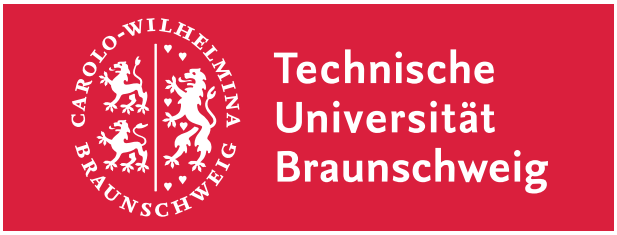


Master's Thesis

Automated Comparison of Product Sampling Algorithms

Author:

Joshua Sprey

August 4, 2020

Advisors:

Prof. Ina Schaefer

Technische Universität Braunschweig

Prof. Thomas Thüm

Universität Ulm

https://doi.org/10.24355/dbbs.084-202009211318-0

Sprey, Joshua:
Automated Comparison of Product Sampling Algorithms
Master's Thesis, TU Braunschweig, 2020.

https://doi.org/10.24355/dbbs.084-202009211318-0

Abstract

The variability of highly configurable systems introduces new challenges and requires new and flexible testing strategies to ensure their quality. Product sampling proved to be one of the most promising strategies to reduce the testing effort by computing a set of products (i.e., a sample) that represents the whole system in the testing phase. A scientific survey about classifying product sampling for software product lines identified more than 48 algorithms to compute samples. We extended the survey with 16 new or missed publications and classified them accordingly to provide a more complete overview. However, the large amount of available sampling algorithms make the user's process to select an appropriate one for their project more complex. Further, the algorithms focus on different objectives (e.g., the runtime of the sampling process or the size of the resulting sample) and there is no complete comparison between all of them. Coupled with the problem of performing redundant evaluations each time a new sampling algorithm is introduced motivated us to design a framework that automatically compares sampling algorithms. Moreover, users often lack the required expert knowledge to understand a complete comparison. This motivated us to provide a strategy to compute recommendations of sampling algorithms that consider the user's requirements. We performed an empirical evaluation on four sampling algorithms with more than 160 real-world systems, including industrial-sized models from the financial services and automotive domain. Based on the data generated by our framework, we concluded that the modern sampling algorithm YASA achieves the best results for multiple objectives. Furthermore, we concluded that our strategy to compute recommendations, named *Weighted Rank-Based Score (WRBS)*, produce correct and precise results.

https://doi.org/10.24355/dbbs.084-202009211318-0

https://doi.org/10.24355/dbbs.084-202009211318-0

Zusammenfassung

Die Variabilität hochgradig konfigurierbarer Systeme bringt neue Herausforderungen mit sich und erfordert neue und flexible Teststrategien zur Sicherung ihrer Qualität. Als eine der vielversprechendsten Strategien zur Verringerung des Testaufwands erwies sich das Produktsamplingverfahren, bei der eine Menge von Produkten (d.h. ein Sample) berechnet wird, welches das gesamte System in der Testphase repräsentiert. Eine wissenschaftliche Umfrage über die Klassifizierung des Produktsamplingverfahrens für Software-Produktlinien identifizierte mehr als 48 Algorithmen zur Berechnung von Samples. In dieser Arbeit wird die Umfrage um 16 neue oder nicht betrachtete Publikationen erweitert, die entsprechend klassifiziert werden, um einen vollständigeren Überblick zu geben. Die große Anzahl verfügbarer Produktsamplingalgorithmen macht jedoch den Prozess des Benutzers komplexer, einen geeigneten Algorithmus für sein Projekt auszuwählen. Zudem konzentrieren sich die Algorithmen auf unterschiedliche Ziele (z.B. die Laufzeit des Produktsamplingverfahrens oder die Größe des resultierenden Samples) und es gibt keinen vollständigen Vergleich zwischen allen Algorithmen. Diese Probleme verbunden mit der Durchführung redundanter Auswertungen bei jeder Einführung eines neuen Produktsamplingalgorithmus motiviert dazu, ein Framework zu entwerfen, welches automatisch Produktsamplingalgorithmen vergleicht. Darüber hinaus fehlt den Anwendern oft das erforderliche Expertenwissen, um einen vollständigen Vergleich zu verstehen. Dies stellt sich als weitere Motivation, um eine Strategie zur Berechnung von Empfehlungen für Produktsamplingalgorithmen zu entwickeln, welche die Anforderungen des Anwenders berücksichtigt. In dieser Arbeit wird eine empirische Evaluation von vier Produktsamplingalgorithmen mit mehr als 160 realen Systemen durchgeführt, darunter Modelle in Industriegröße aus dem Finanzdienstleistungs- und Automobilbereich. Auf der Grundlage der von dem Framework generierten Daten kommen wir zum Schluss, dass der moderne Stichprobenalgorithmus YASA für mehrere Ziele die besten Ergebnisse erzielt. Darüber hinaus wird festgestellt, dass die Strategie zur Berechnung von Empfehlungen, genannt WRBS, korrekte und präzise Ergebnisse liefert.

https://doi.org/10.24355/dbbs.084-202009211318-0

https://doi.org/10.24355/dbbs.084-202009211318-0

Acknowledgements

We would like to express our appreciation to our main advisor Tobias Pett for his continuous support during this thesis. He always took his time to provide constructive feedback for our drafts and ideas. We also would like to thank Dr.-Ing. Thomas Thüm and Sebastian Krieter for their feedback and discussion about several topics of our thesis. We also would like to thank Martin Potthast and the TIRA team for their continuous support and input for our TIRA integration.

https://doi.org/10.24355/dbbs.084-202009211318-0

https://doi.org/10.24355/dbbs.084-202009211318-0

Contents

https://doi.org/10.24355/dbbs.084-202009211318-0

https://doi.org/10.24355/dbbs.084-202009211318-0

List of Figures

https://doi.org/10.24355/dbbs.084-202009211318-0

https://doi.org/10.24355/dbbs.084-202009211318-0

List of Tables

https://doi.org/10.24355/dbbs.084-202009211318-0

https://doi.org/10.24355/dbbs.084-202009211318-0

1. Introduction

Developing large and highly configurable systems requires appropriate engineering techniques. The demand for high-quality products of a shared market segment with individual requirements is high and cannot be fulfilled with traditional approaches. Especially, separately managing every product is time-consuming and often includes redundant effort [LM06]. Feature-oriented product line engineering handles this challenge by providing methods, tools, and processes for developing product lines by reusing engineered assets (e.g., reusable source components, documentation, and more) systematically [KCH+90, LM06, KLD02].

Feature-oriented product lines describe a family of products that share a set of common and varying properties. These properties are distinctive or prominent user-visible aspects, qualities, or characteristics of a system and are identified as *features* [KCH+90]. Product lines provide certain benefits, such as reducing development cost, maintenance effort, and time to market [ABKS13, DMTR08], resulting in many companies adapting their development process to product lines [Wei08]. However, the large amount of possible products aggravates the system's quality assurance as we need to test each product, and thus, traditional testing strategies are infeasible [McG01].

Product sampling is a new testing strategy that reduces the number of products to test by computing a set of products (i.e., a sample) representing the whole system in the testing process [CMMDA12, CDFP97, JHF12a]. The problem is that product sampling is time-consuming and existing algorithms do not scale in terms of CPU time and memory consumption for large-scale systems[MKR+16]. Therefore, product sampling is still researched actively. According to Varshosaz et al. [VAHT+18] more than 40 sampling algorithms to compute samples exist [VAHT+18]. As part of our thesis, we identify new or missing publications and aim to extend the survey to provide an updated overview of product sampling algorithms.

T-wise interaction coverage is a common criterion used as fault coverage in product sampling [VAHT+18]. A t-wise sampling algorithm aims to cover every combination of t features by at least one product of the sample [CMMCDA14, JHF12a, POS+12,

https://doi.org/10.24355/dbbs.084-202009211318-0

OMR10, CMMCDA14]. However, the large amount of available algorithms makes it hard to compare them efficiently, and thus, a complete comparison of all of them does not exist. We discovered that publications about new sampling algorithms manually compare their approach against two to three other algorithms, resulting in a large amount of redundant effort. With our thesis, we aim to provide a platform that automatically compares t-wise sampling algorithms to prevent redundant work in the future and provide an easy way to compare them.

A comparison of sampling algorithms is not sufficient for users in selecting one that suits their needs. Especially as users often lack the insights and time to study all existing sampling algorithms. As the last part of our thesis, we aim to provide a software that uses the comparable data from our platform to recommend the most appropriate sampling algorithm to the user while considering his objectives.

Contribution of this Thesis

We summarize our contribution as follows:

- We extend the survey of Varshosaz et al. [VAHT+18], which classifies product sampling for software product lines by more publications.
- We design a platform that allows users to compare their sampling algorithms against each other automatically.
- We design a software that finds appropriate sampling algorithms while considering the customer's objectives.
- We provide tool support for both our platform and our software.
- We perform an empirical evaluation comparing and finding appropriate sampling algorithms with more than 160 real-world systems, including industrial-sized models.
- We provide a step-by-step guide for our platform, including the setup, evaluation, and comparison of a new sampling algorithm.
- We provide a publicly available repository that contains all of our data, results, and implementations to support our work's reproducibility.

Structure of the Thesis

Our thesis consists of nine chapters. In Chapter 2, we begin with an introduction to the fundamental knowledge to understand our thesis. Then, we motivate our thesis based on two problems we identified in research and industry in Chapter 3. Afterward, in Chapter 4, we introduce the survey of Varshosaz et al. [VAHT+18] and extend their work with more publications to provide an overview of existing sampling algorithms. We present the concepts for our platform to automatically compare sampling algorithms and our software to recommend an appropriate sampling algorithm in Chapter 5. In Chapter 6, we describe our tool support of both concepts. We perform our empirical evaluation with more than 160 real-world systems on both concepts to assess them in Chapter 7. Then, we present the related work for the subject of our thesis in Chapter 8. In Chapter 9, we summarize the most relevant results of our thesis and discuss work that we can address in the future.

https://doi.org/10.24355/dbbs.084-202009211318-0

2. Taming Configurable Systems

This section introduces configurable systems and systematic methods and tools to tame the variability of those systems. We aim to provide the reader with the necessary background knowledge to understand our thesis. In Section 2.1, we introduce feature modeling usage for the variability management of highly configurable systems. In Section 2.3, we introduce the generation of products and product sampling as a solution for assuring the quality of product lines. Then, in Section 2.4, we introduce the sampling algorithm that we use in our thesis.

2.1 Feature Modeling

Feature Model

One of the most common notations for feature-oriented product line engineering is *feature modeling* [BRN+13, FD12, KCH+90]. A feature model describes the features of a product line and their relationship with each other. The aim is to capture and analyze the product line's domain and provide a firm foundation for the communication between different stakeholders devoid of complex source code or component descriptions [ABKS13, CN01, CE00, PBvdL05]. We define feature models based on the definition provided by Knüppel et al. [KTM+18]:

Definition 2.1: Feature Model

A feature model M is given by a 6-tuple $M = (N, r, \omega, \lambda, \Pi, \Psi)$ where :

- *N is the set of features.*
- *r is the root feature of the model.*
- *$\omega : N \mapsto \{0, 1\}$ indicates that a feature $f \in N$ is either mandatory $(\omega(f) = 1)$ or optional $(\omega(f) = 0)$.*
- *$\lambda : N \mapsto \mathbb{N} \times \mathbb{N}$ is a function representing the relationship between a feature and its child features. $\lambda(f) = \langle n, m \rangle$ implies that at least n and at most m children have to be included.*

https://doi.org/10.24355/dbbs.084-202009211318-0

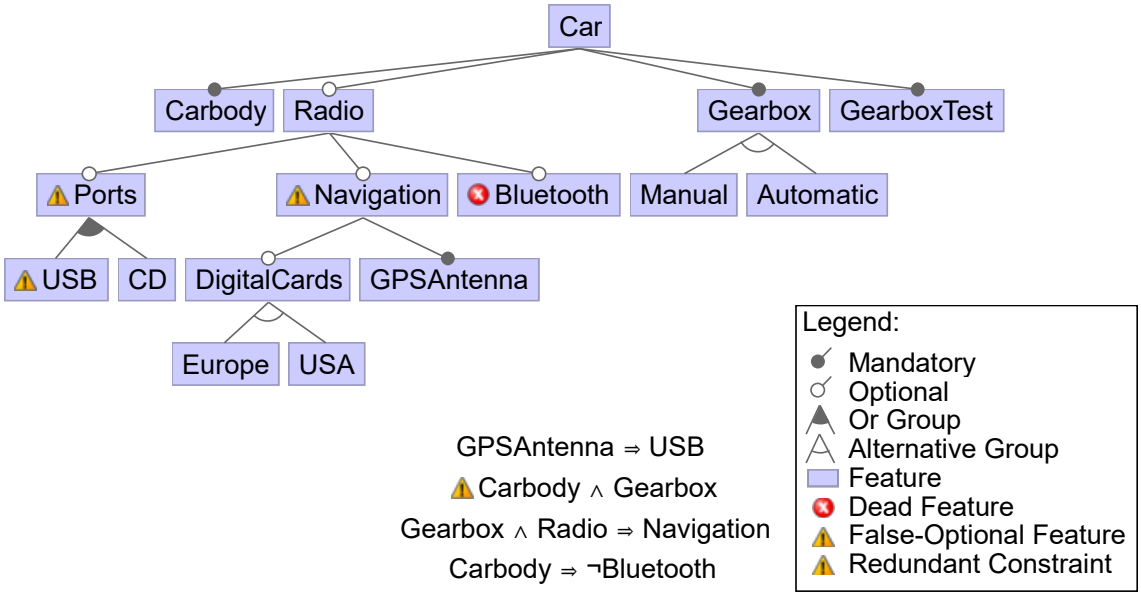


Figure 2.1: Feature model of a simplified car product line

- $\Pi \subset N \times N$ *indicates the parents for each feature as if* $(f, g) \in \Pi$, f *is the parent of* g.
- $\Psi \subseteq Prop_N$ *is the set of cross-tree constraints. We define* $Prop_N$ *as the set of propositional formulas that can be built by using set of features* N *as literals.*

A feature diagram is often used to visualize a product line as tree structure [ABKS13, CN01, CE00, PBvdL05]. For instance, Figure 2.1 shows the feature diagram of a simplified car product line. We explain the syntax of the feature diagram in the following using the above feature model as reference:

Tree-Structures

Feature	References a feature of the feature model [ABKS13].
Connection	Describes a relationship between two different features [ABKS13].

Feature Properties

Optional	The inclusion of the parent does not force the inclusion of an optional child [ABKS13]. For example, a radio is not necessary for a car of our product line, and, thus, the feature `Radio` is optional.
Mandatory	The inclusion of the parent demands the inclusion of all its mandatory children [ABKS13]. For instance, every car requires a car body, and, thus, the feature `Carbody` is a mandatory child of the root feature.

Relationship Types

https://doi.org/10.24355/dbbs.084-202009211318-0

∧	And-Group	Any number of children can be included ($\lambda(f) = \langle 0, n \rangle$) [ABKS13]. For example, we can select a combination of the features `Ports`, `Navigation`, and `Bluetooth` to extend our radio or none at all. Only children of an **And**-group can be **Optional** or **Mandatory**.
▲	Or-Group	At least one of the children has to be included if the parent is included ($\lambda(f) = \langle 1, n \rangle$) [ABKS13]. In our example, we can extend the radio with different ports such as USB and CD. At least one of them has to be selected, and, thus, the features `USB` and `CD` are modeled as an **Or**-group.
⩓	Alternative-Group	Exactly one child has to be included if the parent is included ($\lambda(f) = \langle 1, 1 \rangle$) [ABKS13]. For example, our car requires precisely one of multiples gearboxes as hybrid cars are not covered in our product line. Therefore, we modeled the features `Manual` and `Automatic` as an **Alternative**-group.

Cross-Tree Constraints

`GPSAntenna` ⇒ `USB`	Cross-tree constraints are used to express relations between features from different subtrees. They are constructed as propositional formulas over the features of the feature model [ABKS13]. The example constraint on the left express that having a GPS antenna in our car also requires a USB port.

Cross-tree constraints extend the expressiveness of feature models but also increase the complexity. Furthermore, defects can occur when developing feature models with cross-tree constraints. More than 30 methods in the area of the automated analysis of feature models were introduced to countermeasure the additional complexity [BSRC10]. In the following, we shortly introduce some of the common defects for feature models:

Defects

⛔	Dead Feature	A feature is *dead* if it is not included in any product [KAT16a, BSRC10, ABKS13]. For instance, the constraint `Carbody` ⇒ ¬`Bluetooth` makes the feature `Bluetooth` dead as each car requires a carbody, and, thus, never has a radio with Bluetooth.
⚠	False-Optional Feature	A feature is *false-optional* if it is modeled as optional but has a mandatory relationship to its parent due to cross-tree constraints [NMS+18]. For example, the constraint `Gearbox` ∧ `Radio` ⇒ `Navigation` makes `Navigation` false-optional as the inclusion of a radio always demands the inclusion of `Navigation`.

https://doi.org/10.24355/dbbs.084-202009211318-0

⚠ Redundant Constraint

A constraint is *redundant* if its' semantic information is already modeled by the feature model structure or other constraints [BSRC10, vdML04]. In our feature model, the constraint `Carbody`$\wedge$`Gearbox` is redundant as both features are already mandatory children of the root feature.

2.2 Configuration

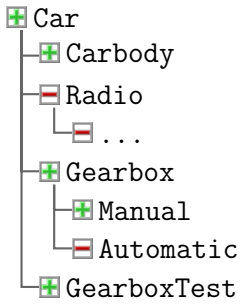
A feature model spans a space of possible products. Users can configure their products by selecting features. This process results in a set of selected and unselected features that characterize this product [ABKS13, HPP+14]. These sets are also called configurations and we define them formally as follows:

Definition 2.2: Configuration

A configuration is a 3-tuple $C = (M, S, U)$ *where:*

- M *is the corresponding feature model.*
- $S \subseteq N_M$ *is the set of selected features.*
- $U \subseteq N_M$ *is the set of unselected features.*
- $S \cap U = \emptyset$

Figure 2.2 shows a visualization for an example configuration on the left side based on our feature model of Figure 2.1. Every feature marked with ⊞ are selected, and every feature marked with ⊟ are unselected for the configuration. Our resulting car would have a manual gearbox, a car body, and no radio features. Furthermore, we show a formal definition and an abbreviated form for the same configuration on the right side of the figure. The logical negate operator ($\neg$) indicates an unselected feature, while all features without negating operator are selected. Most of the time, we use the abbreviated form for the following chapters of this thesis.



$C = (M, S, U)$ with:
M = feature model of Figure 2.1
$S = \{Car, Carbody, Gearbox, Manual, GearboxTest\}$
$U = \{Radio, Ports, ..., Bluetooth, Automatic\}$

Abbreviated form:
C = {Car, Carbody, ¬Radio, ¬Ports, ..., ¬Bluetooth, Gearbox, Manual, ¬Automatic, GearboxTest }

Figure 2.2: A configuration has many representations. The visual representation (left) is one of the most common illustrations, as it is easy to understand. We use the formal text-based definition and its abbreviated form (right) in our thesis as they save space

https://doi.org/10.24355/dbbs.084-202009211318-0

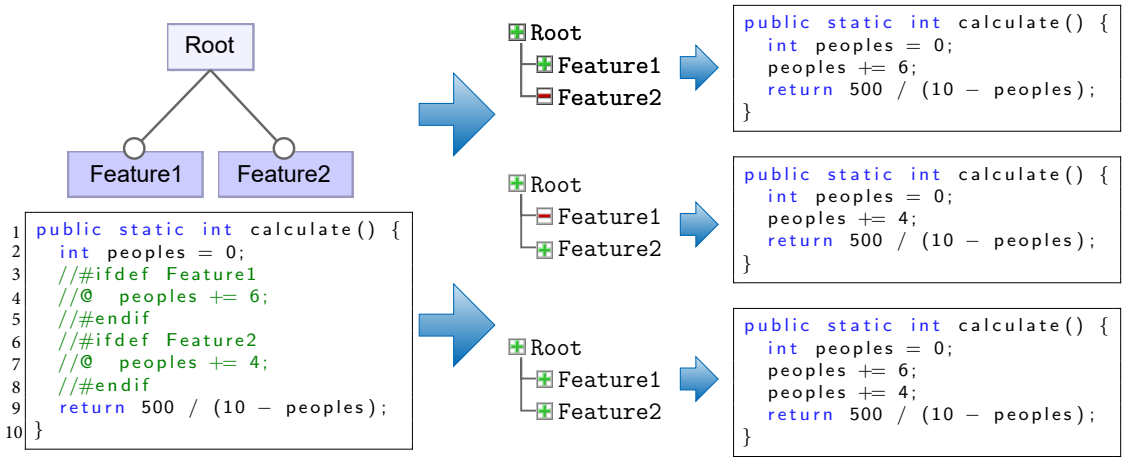


Figure 2.3: Composing three products (right) of a given ANTENNA preprocessor code (bottom left) based on a feature model (top left). Each product is characterized by their respective configuration (middle column)

Valid Configuration

Not every created configuration is automatically a valid configuration (i.e., a product). For example, the following configuration is not valid:

C = {Car, Carbody, ¬Radio, ¬Ports, ..., ¬Bluetooth, Gearbox, Manual, Automatic, GearboxTest }

That is because the features `Manual` and `Automatic` are both selected. However, they are part of an Alternative-group which enforces that only one child feature of `Gearbox` can be selected (i.e., $\lambda(Gearbox) = \langle 1, 1 \rangle$).

The analysis of configurations (e.g., detecting invalid configurations) or feature models (e.g., detecting dead features) is traditionally assisted by solving satisfiability problems (SAT). The feature model and the configuration are transformed into propositional formulas, which are then analyzed with the help of SAT solvers [Man02, BRCTS06, ABKS13]. With that, we can define valid configurations as follows:

Definition 2.3: Valid Configuration

A configuration $C = (M, S, U)$ with feature model $M = (N, r, \omega, \lambda, \Pi, \Psi)$ is called valid, when the propositional formula

$$CNF_M \wedge \bigwedge_{s \in S} s \wedge \bigwedge_{u \in U} \neg u$$

is satisfiable. CNF_M is the feature model's propositional formula in conjunctive normal form.

2.3 Automated Product Generation

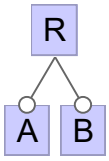
The essential benefit of product lines is their ability to generate individual products based on distinctive configurations and reusable artifacts [ABKS13]. For instance, in Figure 2.3, we show the generation of three products. We provide a feature model with the two features *Feature1* and *Feature2* on the left side and an implementation for a function called `calculate()` as Antenna [PYW11] preprocessor code. The function computes how much money each participant receives. The fund is set to 500 and is divided based on the number of participants. Each feature reduces the number of participants, and thus, has an influence on the result.

We implemented the described behavior of `calculate()` as preprocessor code (cf. bottom left source code in Figure 2.3). We specify that the number of people increases by 6 (cf. line 4) or 4 (cf. line 7) whenever *Feature1* or *Feature2* are selected. Now, we can generate products with configurations. For that, we execute the preprocessor on our implementation along a configuration. The configuration specifies which preprocessor blocks are included and which are removed from the final product. For instance, in the topmost configuration in Figure 2.3, *Feature1* is selected, and therefore, the preprocessor block in line 3-5 is included in the final product. However, *Feature2* is deselected, and thus, the respective preprocessor block in line 6-8 is omitted. This results in the product that reduces the number of participants by 6. The resulting product for the second topmost configuration in Figure 2.3 reduces the number of participants by 4, while the bottom-most configuration reduces the number of participants by 10. Annotation-based approaches are not the only option to compose products for product lines. Many other strategies are used, such as feature-oriented programming, aspect-oriented programming, or runtime variability [MTS+17].

The variability of product lines also comes with new problems to solve. Even today, quality assurance is one of the biggest challenges for product lines [MKR+16]. Assuring the quality of the product lines means assuring the quality of each individual product. For instance, to assure the quality of our feature model in Figure 2.3, we need to compose and test each configuration. Composing products is not sufficient as some faults only appear when executing the product. In our example, we composed all possibles products, and they seem fine for the compiler. However, our bottom-most product results in dividing 500 by zero, which leads to a runtime error. Hence, to ensure the quality, we need to test each product. However, the potential number of products is exponential to the number of features, and thus, traditional approaches are infeasible [McG01].

Product sampling is the process to compute a subset of all products, also referred to as *sample*, that represents our configurable system. Product sampling aims to reduce the effort required by testing or analyzing the configurable system by performing tests and analyses on the samples instead of all valid products. However, computing samples is an NP-complete problem, and many recent works showed that modern sampling techniques in terms of CPU time and memory consumption do not scale for large configurable systems [MKR+16, JHF12a, HPP+12, HNA+19]. Still, sampling is an essential method for various applications in product lines such as testing [VAHT+18, OGB19a], computing statistics [PAP+19, OGB+19b, KGS+19],

https://doi.org/10.24355/dbbs.084-202009211318-0



Feature-Wise (t=1)

Sample	R	A	¬A	B	¬B
C_1	✓	✓	-	✓	-
C_2	✓	-	✓	-	✓

Pair-Wise (t=2)

Sample	(R,A)	(R,¬A)	(R,B)	(R,¬B)	(A,B)	(A,¬B)	(¬A, B)	(¬A,¬B)
C_1	✓	-	✓	-	✓	-	-	-
C_2	-	✓	-	✓	-	-	-	✓
C_3	✓	-	-	✓	-	✓	-	-
C_2	-	✓	✓	-	-	-	✓	-

Figure 2.4: Selecting appropriate interaction coverage for product sampling is important as it influences other criteria. We calculated samples for the feature-wise (t=1) and pair-wise (t=2) coverage of the small feature model given at the top left. We can observe increasing sample sizes and computation effort for growing t values

or predicting configurations with high performance [KGS+19, PAMJ20]. A product sampling algorithm is a piece of software that requires a feature model and other relevant data as input and computes a samples for our product line.

Combinatorial interaction testing (CIT) is one of the most promising strategies to reduce the number of products to test [CMMCDA14, JHF12a, POS+12, OMR10, CMMCDA14]. The fundamental idea is that product faults originate from a combination of absent and present features, also called a *feature interaction*. Therefore, CIT computes a sample that covers a certain degree of interactions. T-wise CIT covers every combination of t features by at least one product of the sample. Figure 2.4 shows samples for both feature-wise (t=1) and pair-wise (t=2) CIT. The feature-wise coverage (upper table) aims to have every feature at least once selected (present) and unselected (absent). The pair-wise coverage (lower table) aims to cover every possible combination of two features at least once. The tables in our example contain the sample on the most left column, consisting of the individual products. The other columns show the conditions for our focused coverage. A checkmark (✓) indicates that the condition of the header is covered in this product. Complete coverage is achieved if every column of a table has at least one checkmark. Invalid conditions were omitted from the tables (e.g., the root feature cannot be unselected (¬R)). Our sample for feature-wise coverage consists of two products, while the sample for pair-wise coverage consists of four products. Both achieve complete coverage. A sample that achieves a complete t-wise coverage also implies a complete coverage for smaller t values [FCP09].

2.4 Sampling Algorithms

In this subsection, we introduce the algorithms that we use for our thesis.

Chvatal

T-wise product sampling can be adapted to a minimum set-covering problem, i.e., a well known NP-complete problem:

https://doi.org/10.24355/dbbs.084-202009211318-0

Definition 2.4: Set-Covering Problem

Given a set of elements U and a set of sets $S = \{s_1, s_2, ..., s_n\}$ with $s_i \subseteq U$ for $0 \leq i \leq n$ and any $n \in \mathbb{N}$. Find the minimum number of sets $C = \{c_1, c_2, ..., c_k\} \subseteq S$ such that:

$$\bigcup_{i=0}^{k} c_i = U$$

In 1979, Chvatal published an algorithm to generate covering arrays for any strength, i.e., the degree of interactions between sets. The resulting covering arrays always reach a 100% coverage of interactions, but they are not minimal, and thus, the algorithm is a greedy heuristic [Chv79]. Johansen et al. adapted Chvatal's algorithm for the generation of t-wise samples of feature models [JHF11]. In the following, we show a small example of an adaption of feature-wise product sampling to the set-covering array and the usage of the CHVATAL algorithm.

Example 2.5: Set-Covering Problem for Feature-Wise Sampling

We compute feature-wise samples for the model of Figure 2.4 with the CHVATAL algorithm. We identify the feature-wise sampling process as the following set-covering problem:

Assume S is the set containing all products. Given U contains all conditions of interest for our feature-wise coverage. Again ¬R is omitted for brevity.

$U = \{< R, true >, < A, true >, < A, false >, < B, true >, < B, false >\}$

The algorithm is quite simple. We create a new configuration and add uncovered feature combinations until the configuration becomes invalid. The last combination is removed, and the final configuration is added to C. We repeat until all conditions are covered. The same procedure can be used for any value of t. Given the feature model M, one element of U would have the following form:

$U = \{x | x =< f_1, b_1, f_2, b_2, ..., f_t, b_t >\}$ with $f_i \in N_M$; $b_i \in \{true, false\}$; $t \in \mathbb{N}$

ICPL

Johansen et al. created ICPL based on their adaption of Chvatals algorithm [JHF12a]. Their first extension improves the detection and handling of invalid feature combinations. Utilizing efficient analyses for the feature model domain allows ICPL to detect more invalid combinations early on and helps to prevent redundant work. Also, some subroutines are performed in parallel to further improve the scalability of the algorithm and his usage for large feature models.

https://doi.org/10.24355/dbbs.084-202009211318-0

IncLing

Small sample sizes are essential as they reduce the test effort. However, computing the sample also requires a lot of time and resources. Al-Hajjaji et al. created IncLing, a pair-wise sampling algorithm based on ICPL, to reduce the transition time between sampling and testing by computing samples iteratively [AHKT+16]. By providing intermediate samples after every iteration, it is possible to start the testing process in parallel. Invalid feature combinations are detected before the actual sampling process starts at the cost of more computation time to prevent invalid intermediate samples. This validation ensures that the testing process does not test invalid products. Further improvements in the area of the automated domain analysis, in contrast to ICPL, also improved the overall process of IncLing.

YASA

T-Wise interaction testing for large systems with a traditional sampling algorithm is limited [MFBW16, JHF12a]. YASA wants to address the scalability problem of large-scale systems by introducing a new approach for t-wise testing with the focus on scalability and flexibility [KTS+20]. For this, YASA introduces three improvements in contrast to general algorithms. First, instead of covering all t-wise feature interactions, only a customized subset is covered. Choosing customized sets of feature interactions with available domain knowledge can increase the efficiency of the sampling process. Second, using heuristics, caching, and pre-computed data improves the performance. Third, providing parameters to fine tune, the sampling process lets the user control the trade-off between sampling time and sample size. YASAs sampling process is different from the Chvatal algorithm and its derivations. The general process consists of iterating all feature interactions of interest and adding them to a new or existing configuration until all interactions are covered.

https://doi.org/10.24355/dbbs.084-202009211318-0

https://doi.org/10.24355/dbbs.084-202009211318-0

3. Problem Statement

In this chapter, we motivate the two challenges of our master thesis. In Section 3.1, we introduce the first challenge that research is more difficult due to redundant work and missing elements of reproducibility in publications. In Section 3.2, we introduce the second challenge addressing the selection of an appropriate sampling algorithm for individual requirements.

3.1 Uncoordinated Shared Task

Scientific work is an essential process for finding new knowledge and technologies. Computer science, in particular, is an effective area of research as all reproducibility elements are digitally available. In general, every scientific publication is supported by an experiment consisting of software and data. These elements can be easily packed and transferred over the Internet. This can increase the credibility and efficiency of scientific work as it can be easily reproduced and verified by other researchers of the same area [PGWS19].

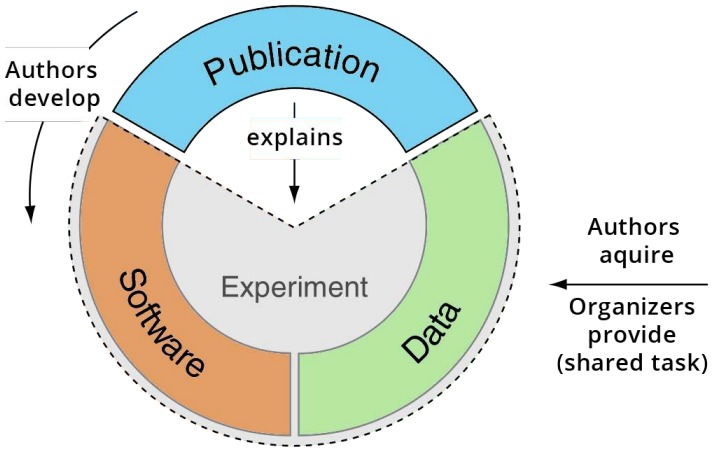


Figure 3.1: Shows elements for a coordinated shared task that realizes easily reproducible results. Figure designed by Potthast et al. [PGWS19] and adapted by us

https://doi.org/10.24355/dbbs.084-202009211318-0

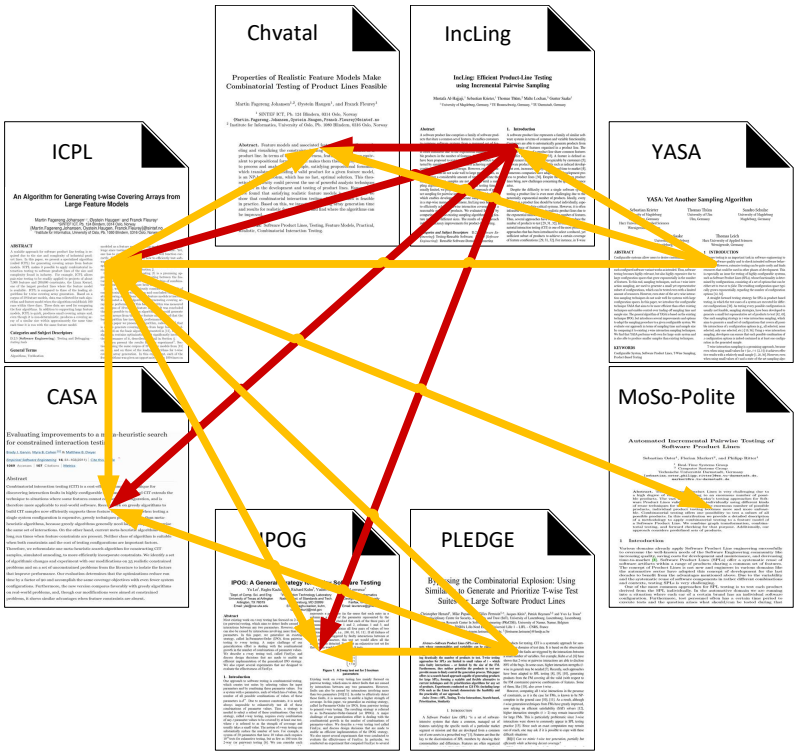
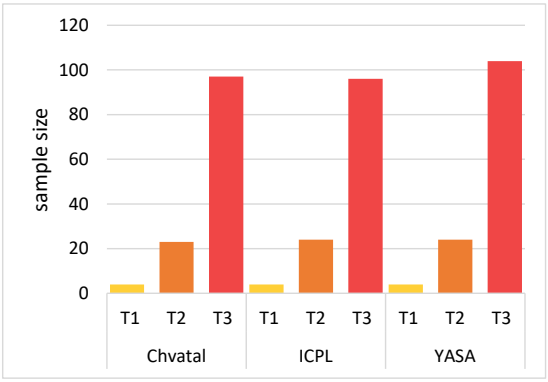


Figure 3.2: We identified the relations between a set of research papers for sampling algorithms depending on the amount of redundant work performed by the authors. Outgoing yellow arrows indicate that the author evaluated the targeted article. Also, a red arrow extends a yellow arrow and show that the author also reimplemented the targeted algorithm

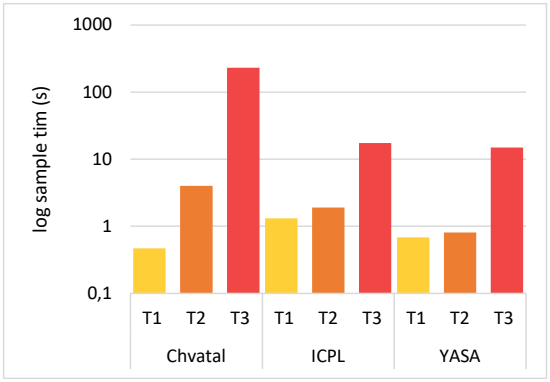
It is common for researchers to work on the same tasks. In Figure 3.1, we show the three main elements needed to reproduce scientific work in computer science. The author of the publication is responsible for writing his software and explaining the experiment. Ideally, in a coordinated shared task data can be retrieved from an organizer. This could help as other researchers can perform their evaluation with the same set of data, and, thus, making their results easier to compare and to reproduce. However, in general, shared tasks are often uncoordinated and result in redundant work. Furthermore, it is not natural to share software and data with peers, and it is often necessary to reimplement the software of other authors or reevaluate their approaches with a different set of data.

We can observe this challenge for the product sampling community. In Figure 3.2, we show eight publications for product sampling algorithms and their relations in context to our challenge. We cover the following algorithms: CHVATAL [Chv79], ICPL [JHF12a], INCLING [AHKT+16], CASA [GCD11], IPOG [LKK+07], YASA [KTS+20], MOSO-POLITE [OMR10], and PLEDGE [HPP+14]. These are only a subset of all available algorithms but still sufficient to show the challenge. It is common in the community to compare a new algorithm against other algorithms. We mark such relations in the form of arrows. For instance, YASA compares against INCLING, CHVATAL, and ICPL, and, therefore, we have three outgoing arrows from YASA to the three algorithms. Further, we differentiate between two kinds of re-

https://doi.org/10.24355/dbbs.084-202009211318-0



(a) Shows the sample size of the sampling algorithms CHVATAL, ICPL, and YASA for the t-values 1-3



(b) Shows the runtime in logarithmic scale of the sampling algorithms CHVATAL, ICPL, and YASA for the t-values 1-3

Figure 3.3: We calculated the testing efficiency (left) and sampling efficiency (right) for increasing t values to show the tension between coverage and efficiency of product sampling algorithms

lationships. The red arrows indicate the necessity to reimplement and to reevaluate the other algorithms while the yellow arrows indicate only to reevaluate the other algorithms.

The trend for newer algorithms such as INCLING and YASA could indicate that new sampling algorithms compare their results against three to four other algorithms on average. This results in many redundant evaluations. For instance, INCLING, ICPL, and YASA perform an individual evaluation of the CHVATAL algorithm with a different set of data. Instead of evaluating every algorithm all over again, we strive to establish product sampling as a coordinated shared task to prevent these challenges. In Section 5.1, we propose our concept to realize product sampling as a coordinated shared task.

3.2 Selecting Appropriate Sampling Algorithms

The sampling process is essential for the industry to ensure the quality of their systems, especially for safety-critical systems. Varshosaz et al. [VAHT+18] identified around 38 publications for sampling algorithms, and each algorithm has advantages and drawbacks. Hence, selecting an appropriate algorithm is not an easy task, especially as the whole shared task for product sampling is uncoordinated. Furthermore, the requirements for testing efficiency, sampling efficiency, and feature interaction coverage can be highly diverse for every user. For instance, user A has a system where testing one product is complicated and time-consuming, while the computation of the sample has no further constraints. So user A wants a higher testing efficiency (i.e., smaller sample size) for the sake of a lower sample efficiency (i.e., high sample computation time).

There exists a tension triangle between sampling efficiency, testing efficiency, and feature interaction coverage, and, thus, there is not a single sampling algorithm that performs best for all criteria at the same time. For example, in Figure 3.3, we

https://doi.org/10.24355/dbbs.084-202009211318-0

show the results for the calculation of feature-wise (t=1), pair-wise (t=2), and t=3 samples. The left graph shows the resulting sample sizes, while the right graph shows the required sample times. We performed the calculations on the BerkeleyDB model[1] provided by FeatureIDE [MTS+17] for the sampling algorithms Chvatal [Chv79], ICPL [JHF12a], and YASA [KTS+20]. We see that a higher feature interaction coverage leads to a lower sampling efficiency and higher testing efficiency. These calculations indicate that selecting an appropriate algorithm is essential to fulfill the user's requirements. The problem is the missing expert knowledge of the industrial user in selecting an algorithm as each algorithm focuses on different objectives, i.e., the emphasis of sampling criteria. In Section 5.2, we introduce a concept to select an appropriate algorithm while considering individual requirements of sampling criteria.

[1] https://github.com/FeatureIDE/FeatureIDE/blob/develop/plugins/de.ovgu.featureide.examples/featureide_examples/BerkeleyDB-FH-Java/model.xml

https://doi.org/10.24355/dbbs.084-202009211318-0

4. Survey on Product Sampling

In this chapter, we focus on introducing and extending a survey for product sampling to provide an overview of the current research area and further motivate our thesis. We begin in Section 4.1 with an introduction to the classification survey of product sampling. Then, in Section 4.2, we extend the classification survey by identifying ten research papers. In Section 4.3, we summarize the most important aspects of this chapter.

4.1 Product Sampling Classification

In 2018, Varshosaz et al. [VAHT+18] published a classification of product sampling for software product lines. They focused on papers that introduced new sampling algorithms or evaluated existing ones. All in all, they identified 48 works and classified them based on the input data for algorithms, the type of algorithm used, and their evaluation. In Figure 4.1, we show a mind map containing the categories of Varshosaz et al. classification. In the following, we shortly explain the different (sub)categories:

4.1.1 Category: Input Data

The category of input data encompasses every kind of data that was given to a specific sampling algorithm. The various input data (cf. blue category in Figure 4.1) was classified into *feature models*, *expert knowledge*, *implementation artifacts*, and *testing artifacts*. First, **feature models** were generally used to distinguish valid from invalid configurations. Varshosaz et al. [VAHT+18] showed that most strategies use feature models to detect invalid product when computing the samples. Second, **expert knowledge** can be used to define a set of manually selected products that are given into the sampling process. For example, the developer especially selects products containing features that mostly produce errors based on his knowledge. Sampling techniques by Oster et al. [OMR10] or Al-Hajjaji et al. [AHKT+16] compute samples based on an optionally set of predefined products. Third, some sampling processes consider solution space information and require the respective

https://doi.org/10.24355/dbbs.084-202009211318-0

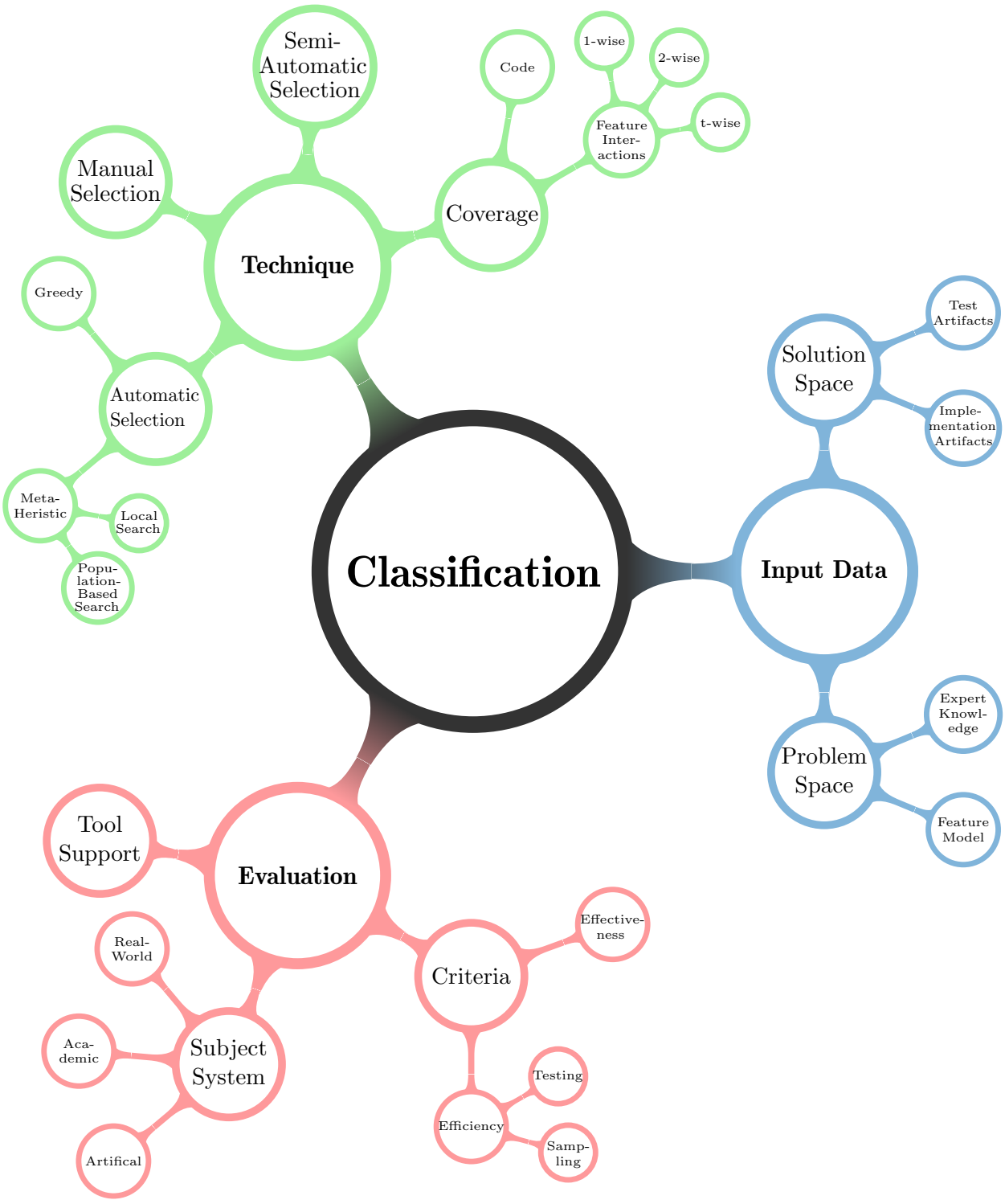


Figure 4.1: Varshosaz et al. [VAHT+18] classified product sampling by three categories. The mind map shows the three categories: *Technique (green)*, *Input Data (blue)*, and *Evaluation (red)* and their subcategories

https://doi.org/10.24355/dbbs.084-202009211318-0

implementation artifacts. For example, Tartler et al. [TLD+12] proposed a sampling strategy for preprocessors and requires code containing preprocessor directives as input. Fourth, we can use **testing artifacts** such as unit or runtime tests to detect which features influence the results of these tests.

4.1.2 Category: Techniques

The category of techniques contains the different principal approaches of computing sample sets. We differentiate between *manual selection*, *semi-automatic selection*, *automatic selection*, and *coverage*-based techniques. We marked all related classification categories for the techniques green in Figure 4.1.

Manual selection techniques allow domain experts to select the set of products manually based on their knowledge about the subject system.

Semi-Automatic selection techniques run automatically but are constrained by different types of data such as the maximum time used for sampling, the required number of products to generate, or a degree of coverage to achieve. Further, semi-automatic techniques can use the full or partial sample sets computed by other techniques as a starting point for sampling.

Automatic selection techniques are differentiated into *greedy* and *meta-heuristic* techniques. **Greedy** techniques compute sample sets by choosing a locally optimal choice in each stage. Each choice is a new product that gets added to the sample set. **Meta-heuristic search** techniques use computational search in the configuration space to select a subset of products as an optimal solution for product sampling. We differentiate meta-heuristic search techniques into *local search* and *population-based search*. **Local search** techniques aim to compute a near-optimal solution by gradually evolving a starting preliminary set of products. Examples for such techniques are simulated annealing and tabu search. **Population-Based search** approaches start with a preliminary set of sample sets (sets of products). They mutate and recombine the starting sets among each other until they find a near optimal solution. Examples of population-based approaches are genetic algorithms and swarm techniques.

Coverage-based techniques use criteria to estimate the quality of product sampling. They are differentiated into *code* coverage-based and *feature interaction* coverage-based techniques. **Code** coverage-based techniques compute samples to cover a certain percentage of the code, while feature interaction-based approaches aim to cover all t-wise feature interactions for a specific degree.

Feature interaction-based techniques compute a sample to provide complete coverage of feature interactions to a certain degree (t-value).

4.1.3 Category: Evaluation

The category of evaluation contains the most important factors for performed evaluations. For the evaluation, we differ between *tool support*, *subject system*, and *criteria*. We marked all related classifications categories for the evaluation red in Figure 4.1.

https://doi.org/10.24355/dbbs.084-202009211318-0

Tool support indicates whether an implementation supports a sampling technique. Further important factors are the availability of such tools and whether they are open or closed source.

Subject system describe the scalability and practicality of a technique. For that, Varshosaz et al. [VAHT+18] differentiate between ***artificial***, ***academic***, and ***real-world*** systems.

Criteria, or evaluation criteria, are used to compare techniques based on their coverage, performance when sampling, and performance when testing the resulting samples. They further categorized into *efficiency* and *effectiveness*.

Efficiency estimates the physical performance of product sampling techniques. The authors divide the property into *sampling efficiency* and *testing efficiency*. **Sampling efficiency** considers the time and resources used to generate the sample [VAHT+18]. For our thesis, we will separate the algorithm's *runtime* and *memory consumption* by considering them as individual criteria for our evaluation. **Testing efficiency** focuses on the properties of the resulting sample, such as the sample's size and the size of the individual products [VAHT+18]. For our thesis, we focus on the *sample size* as a criterion.

Effectiveness is an indication of the quality of the sample. The quality of a sample can decided by reaching a certain *feature interaction coverage*, having a high *fault coverage*, achieving a set *code coverage*, or computing stable samples over time (*sample stability*). **Fault coverage** is the capability of measuring faults in the subject systems. **Code coverage** is the percentage of code in the solution space that is covered by a generated sample. For our thesis, we do not cover the solution space for our systems, and thus, we do not consider fault/code coverages. **Feature interaction coverage** checks whether the resulting sample of a technique achieves a t-wise interaction coverage [VAHT+18]. The coverage is often used for fault coverage as faults result in interactions between features. For our thesis, we name it *sample coverage* for the following chapters. **Sample stability** describes how much a sample changes over the product-line evolution [Pet18a]. Pett [Pet18a] introduces several metrics to assess the stability of a sample. To compute all metrics, we require a sample before and after an evolution:

Ratio of Identical Configurations (RoIC) calculates the similarity between the two samples by comparing the number of their identical configurations.

Mean Similarity of Configurations (MSoC) uses a heuristic to match the most similar configurations and calculate the sample similarity by aggregating the similarity of the matched configurations. However, the heuristics do not always match configurations perfectly.

Filter Identical Match Different Configurations (FIMDC) is based on MSoC and improves the heuristic by introducing a preprocessing to match identical configurations.

For our thesis, we consider the stability of sampling algorithms by measuring the *sample similarity*.

https://doi.org/10.24355/dbbs.084-202009211318-0

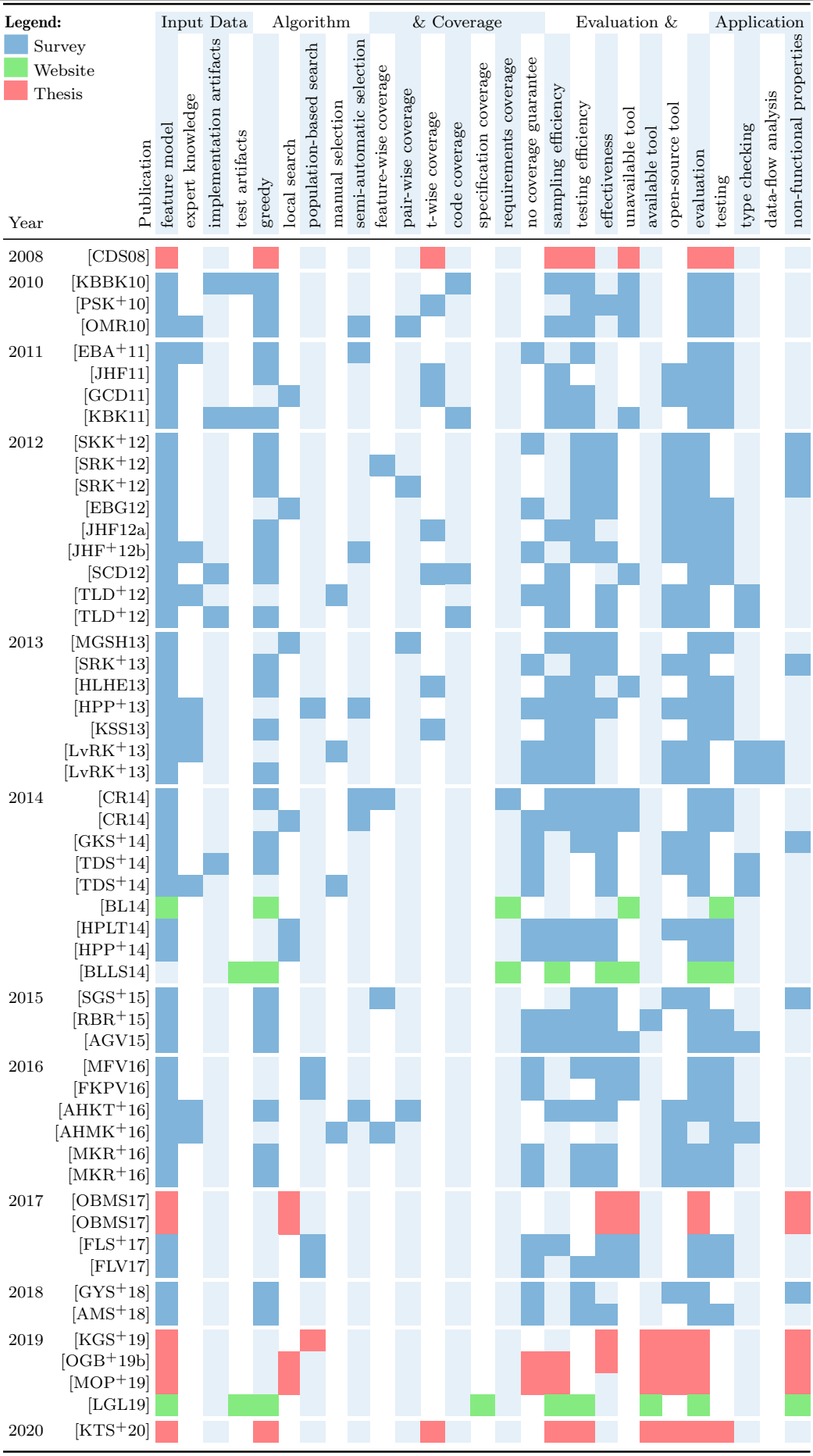


Table 4.1: Overview and extension for the survey of Varshosaz et al. [VAHT+18] for publications introducing new sampling techniques. The legend (top left) shows the color code that describes the origin of each publication. Table designed by Varshosaz et al. [VAHT+18] and extended by us

https://doi.org/10.24355/dbbs.084-202009211318-0

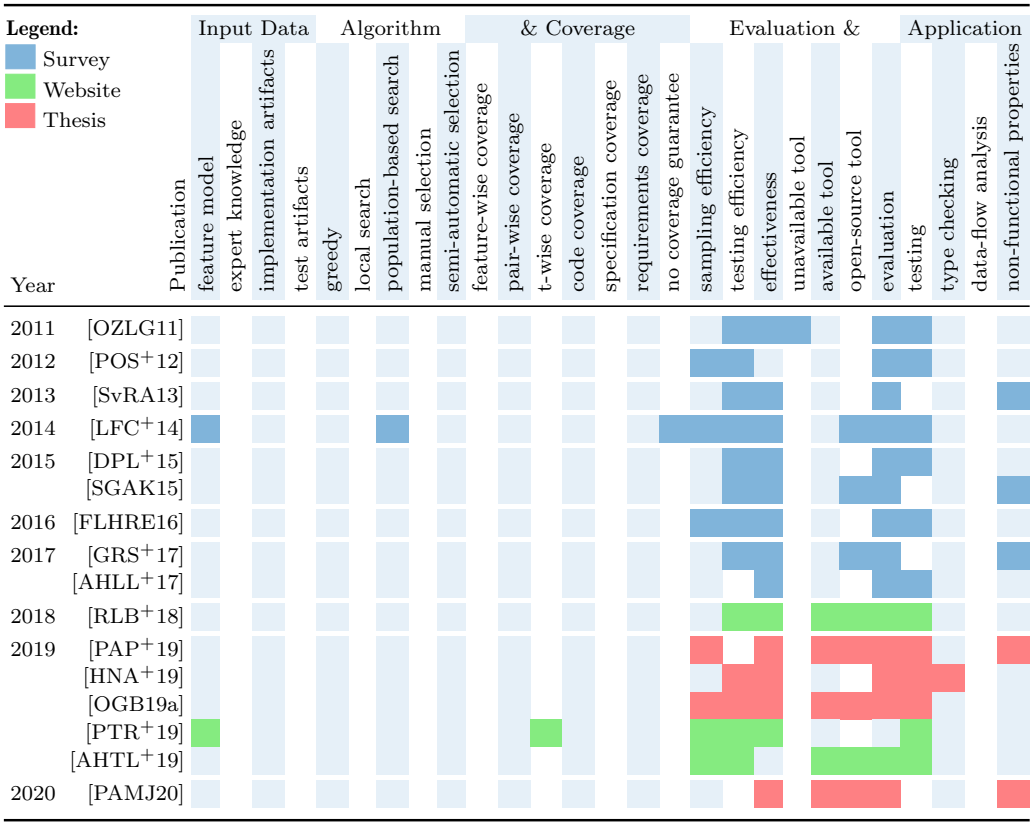


Table 4.2: Overview and extension for the survey of Varshosaz et al. [VAHT+18] for publications that only perform evaluations of existing techniques. The legend (top left) shows the color code that describes the origin of each publication. Table designed by Varshosaz et al. [VAHT+18] and extended by us

4.2 Extending the Survey

The survey is not complete as it misses a fundamental publication. Also, new works concerning product sampling got published in recent years. The authors also published a website[1] for the product sampling survey alongside the paper as an artifact. It contains a filterable and more detailed overview of the individual publication of the survey. Additionally, it is updated from time to time and is more up-to-date as the survey. We aim to extend the classification survey of Varshosaz et al. [VAHT+18]. For that, we verify the classifications of the new publications that are available on the website. Further, we identify ten new publications and classify them accordingly. All publications that we classified are available on our GitHub repository[2] as PDFs, which contain annotations to reason our classification.

We use the same table that was already used by Varshosaz et al. [VAHT+18] for their survey to provide an overview of all already classified publications and to show our results. For the brevity of the resulting table, we group all publications into two groups. The first group contains all publications that introduce new or modify existing sampling techniques. Table 4.1 shows all publications for the first group. The second group contains publications that only evaluate existing techniques. Table 4.2

[1] http://thomas.thüm.de/sampling/
[2] https://github.com/Subaro/Masterthesis-Data/tree/master/Classification

https://doi.org/10.24355/dbbs.084-202009211318-0

shows all publications for the second group. Sometimes a publication introduces several techniques, and each of them deserves an individual classification. In that case, the publication has two or more entries in our table, depending on the number of new techniques. We marked each publication differently in both tables depending on their source:

- Publication was classified by Varshosaz et al. [VAHT+18] in their survey
- Publication was classified on the website and verified by us
- Publication is classified by us

Now, we describe our observations in each classification category for all publications that we classified.

Input data We can observe that all papers require feature models as input for their techniques or evaluations. There are no further types of data used as input.

Techniques We identified two greedy t-wise algorithms: AETG [CDS08], a family of greedy CIT algorithms, and YASA [KTS+20], a configurable algorithm. The configuration for YASA determines the trade-off between the runtime and the size of the resulting sample. It is, therefore, quite flexible and will be considered further in the course of our thesis. We also identified two population-based approaches that are called distance-based sampling [KGS+19, PAMJ20]. They are generally used in the context of determining the most performant configuration of a configurable system. The four local-search algorithms use the same underlying technique called Uniform Random Sampling (URS) [OBMS17, OGB+19b, OGB19a, PAP+19, PAMJ20]. Oh et al. [OBMS17] introduce URS on feature models to compute uniformly distributed samples among the configuration space. This property is especially helpful for computing representative statistics for product lines. Many studies use URS for the use-case of computing statistics [OBMS17, OGB+19b, PAP+19, PAMJ20]. However, URS can also be used to compute t-wise samples without a coverage guarantee [OGB19a].

Evaluation Sampling efficiency was treated equally between the identified publications. All papers measure the runtime of their technique. Regarding the testing efficiency, only testing-based approaches measure the sample sizes [CDS08, KTS+20, OGB19a, PAP+19]. The effectiveness of the different techniques was considered differently. Testing-based approaches [CDS08, KTS+20, OGB19a, PAP+19] measured the effectiveness in the form of feature interaction coverage. Distance-based sampling publications [PAP+19, PAMJ20] measured the effectiveness based on their prediction of the most performant configuration. Moreover, URS publications [OGB+19b, PAMJ20] measured how uniformly distributed their samples are.

Tool support Varshosaz et al. [VAHT+18] published their classification in 2018, and all of our identified publications, except [CDS08], are from 2017-2020. The recent publications provide a useful description of how to replicate their work and provide a reference to their implementation. This reaction could indicate that researchers perceive the problem of missing evaluation software and data, as described in Section 3.1, and act by proactively sharing research artifacts.

The survey can provide an overview of the different product sampling algorithms and their constraints. However, with our extension, there are already 54 sampling

https://doi.org/10.24355/dbbs.084-202009211318-0

techniques. Moreover, the survey does not provide information about how good these algorithms perform against each other. Additionally, not every user has the time and expert knowledge to read all papers to an understandable extend. All in all, the extended survey and the trend towards proactive sharing of research software are already steps in the right direction. The next step is to automate the comparison of product sampling algorithms.

4.3 Summary

To summarize, we introduced the classification for product sampling and all of the subclassifications: *input data*, *technique*, and *evaluation*. Input data is the kind of data that a sampling technique receives. Sampling techniques describe the different kind of principal strategies to generate samples. Evaluation encompasses all relevant aspects to evaluate a sampling technique computing a sample for a given subject system. We specified that our master thesis focuses on testing product lines and that we only consider *runtime*, *memory consumption*, *sample size*, *sample coverage*, *sample similarity* as criteria for our thesis. Then, we extended the survey by identifying ten new publications and verifying six other works. We identified four new techniques: AETG, YASA, distance-based sampling, and uniform random sampling. We introduced the different techniques and categorized them based on input data, evaluation, and technique.

https://doi.org/10.24355/dbbs.084-202009211318-0

5. Automated Evaluation of Product Sampling

In this chapter, we present our approach to addressing the uncoordinated shared task (cf. Section 3.1) and selecting appropriate sampling algorithms (cf. Section 3.2). We introduce a concept for a platform in Section 5.1, which automatically compares sampling algorithms to transform the current uncoordinated shared task into a coordinated one. Afterward, in Section 5.2, we present a concept to determine the most appropriate sampling algorithm based on individual requirements and the results from our platform. In Section 5.3, we summarize this chapter and the most important information on both concepts.

5.1 Platform for Automated Comparison of Sampling Algorithms

In this section, we introduce a platform to perform automated comparisons of product sampling algorithms. We aim to motivate and support the transition from an uncoordinated shared task into a coordinated one. To do so, we provide a platform where researchers can focus on their algorithms while preventing tedious, redundant work for other algorithms. The main benefit is that researchers have more time to spend on developing their work. In the following, we introduce our concept for such a platform and elaborate on various aspects.

5.1.1 General Concept

In Figure 5.1, we show the general concept of the platform. The three main components are several packages of data, different sampling algorithms, and the publicly available results. Before we describe each component's details, we explain the overall process for a researcher that works on an entirely new sampling algorithm. For instance, on the left in Figure 5.1, the researcher Max is having a great idea of computing samples and calls his sampling algorithm SAVM. With the current standards

https://doi.org/10.24355/dbbs.084-202009211318-0

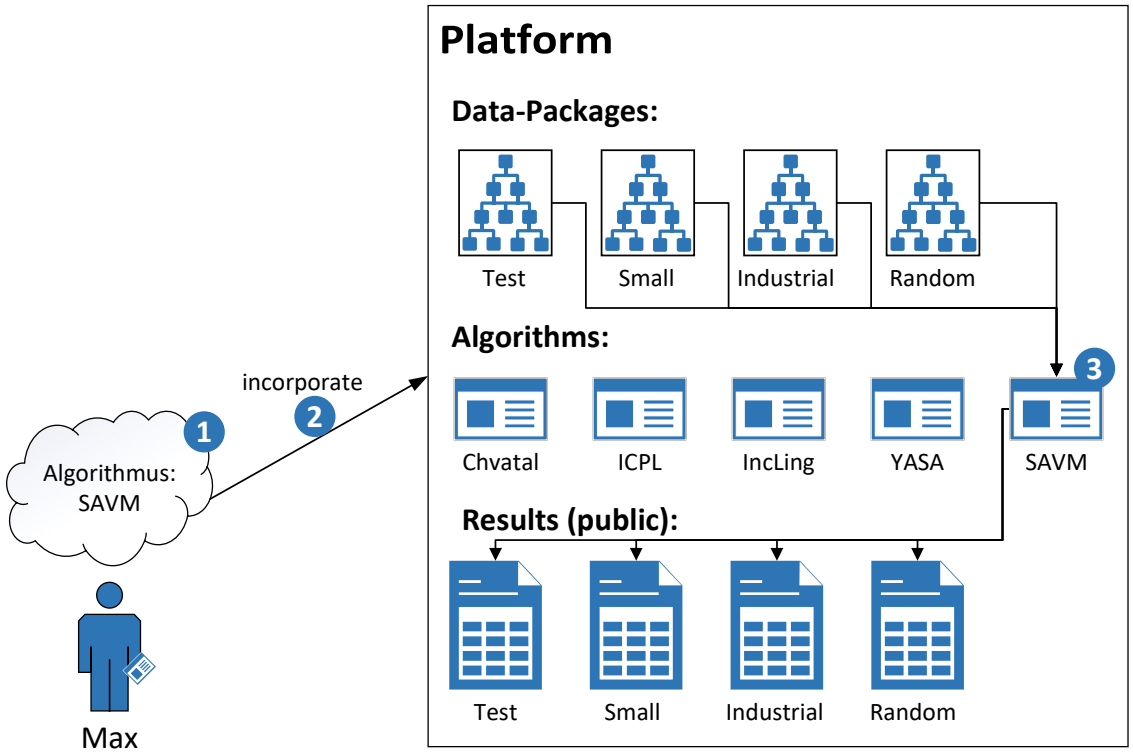


Figure 5.1: The General concept for the platform addressing the problem of researchers working on a new sampling algorithm

of the research area, Max needs to compare his algorithm against at least three other sampling algorithms. That includes time to get the algorithm's software, learning how to use it, and finally evaluating them. To make his results more representative, Max also considers searching for data used to evaluate the other algorithms. However, data is often prepared and not publicly shared, and thus, Max needs to contact the different authors or re-engineer the data. Both processes are time-consuming.

Instead of having to perform this tedious process, we provide our platform to Max. He can use it as follows:

1. In the first place, Max needs to implement his algorithm.
2. Afterward, Max integrates his software into our platform easily via an interface.
3. The platform then performs an automated evaluation with SAVM on all available data packages. The platform automatically stores the results for each data package in the respective result tables.

Overall, the platform provides multiple benefits for Max. First, no preparation for data is necessary. Second, instead of evaluating four algorithms, it is only necessary to perform one evaluation, reducing the theoretical effort by about 75%. Third, the results for all available sampling algorithms are present in the result tables. In the end, Max can extract the results from the respective tables and immediately compare SAVM against all other algorithms.

Not everyone is happy about platforms that automatically perform evaluations as they control the experiment. Doubts about software privacy, the credibility of avail-

https://doi.org/10.24355/dbbs.084-202009211318-0

able data, or exploiting platform mechanics to gain advantages are reasonable. Hence, transparent handling of essential aspects such as data integrity, software integrity, and evaluation credibility is necessary. For that, we propose a list of integrity requirements for our platform. After presenting the requirements, we focus on the components of the platform (i.e., data packages, algorithms, and result tables), describe them, and explain how we realize the requirements. We categorize our integrity requirements in the following two categories:

Data:

- *D1*: Data packages should be expandable.
- *D2*: Data packages are categorized in two categories:
  1. Public packages for testing the implementation that can be accessed by the user at any given time.
  2. Private packages for evaluation that cannot be accessed by the user at any given time.

Software:

- *S1*: Adding new software shall be easy.
- *S2*: The sampling algorithm software shall only be accessed by the user.
- *S3*: Users can experiment and execute their software with testing data without restraints.
- *S4*: All software is executed on machines with the same specifications.

Data-Packages

The first component of our platform is the data. Each data package consists of multiple models for product line systems. Modern sampling algorithms do not scale for large systems, and therefore, it is not efficient to evaluate small, and big industrial models together as results are only available when all models are processed [VAHT+18, MFBW16, MKR+16, HPP+14]. With that in mind, we introduced different data packages, mostly depending on the sizes of our models. Further, it is possible to consider models with a different format, models that were preprocessed by a specific algorithm, or models that are publicly available as criteria for data packages. Such data variety could motivate participants to compete in different contexts. For example, the sampling algorithm that scales best for large systems is not automatically as efficient when processing small systems. Therefore, we provide a diverse range of data packages to observe such a competition.

It is essential to construct the platform so that it is easy to introduce, expand, and remove data packages to keep data up-to-date. For our platform, we design data packages as ZIP files containing multiple models of product lines, and, hence, adding or removing models is quite easy. With that, we fulfill *D1*.

For *D2*, it is necessary to provide data packages to the platform for evaluation while preventing the leak of private data. Therefore, selecting a correct submission type for our platform is important. Figure 5.2 shows three possible types of submissions.

https://doi.org/10.24355/dbbs.084-202009211318-0

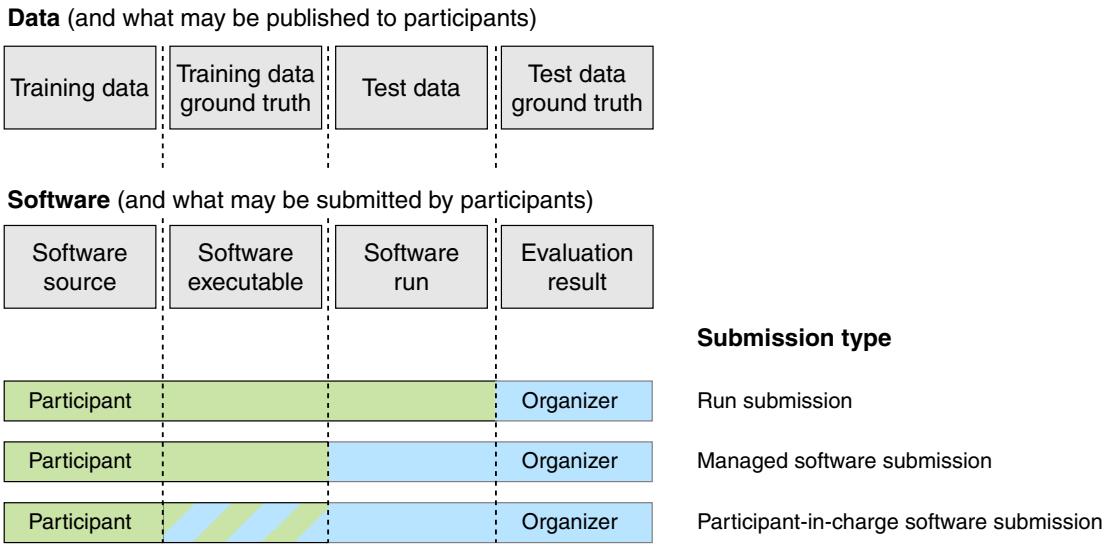


Figure 5.2: From top to bottom: Shows the three types of submission and the responsibility of the organizer and participants towards data and software. For instance, a run submission-based platform gives the participant the responsibility to write the software code, control the execution, and to perform runs. However, the organizer is responsible for evaluating the results. Figure designed by Potthast et al. [PGWS19]

When we compare submission types, we compare the responsibilities for organizers and participants on software and data. For instance, this includes the access to training and test data. Any data that can be accessed by the participant is publicly available. However, all data under the jurisdiction of the organizer is private [PGWS19]. We can exclude run submission as an appropriate type for our platform as the participant has access to the test data, i.e., violates data integrity requirement *D2*. For further exclusions, we need to describe the second component of the platform.

Our second component is the sampling software. When submitting software to a platform, we can split the process into four general steps. The first step is the creation of the software. The second step controls the execution of the software, e.g., executing software in a sandbox with no Internet connection. The third step is the actual execution of the software (i.e., a software run) and the final step is evaluating the run. In managed software submissions, the participants submit executables for the organizer. This process leads to a higher level of reproducibility and automation as an organizer can generate results themselves. Furthermore, it is in accordance with *D2* as an organizer does not need to publish their test data. However, managed software submission are not popular as they include organizer in the debugging process of the software. For instance, user P submits his executable to the organizer O. O executes the software to generate results. However, now some errors appear, and the process is interrupted. O needs to inform P of the error and wait for a fixed submission. This process can repeat and severely increases the time to identify errors, especially when both parties are not working simultaneously. Further, many researchers begin their work shortly before a deadline, and thus, the organizer needs

https://doi.org/10.24355/dbbs.084-202009211318-0

to process all submissions at the same time. The drawbacks often outweigh the benefits for managed software submission, and as a result, it is not popular among organizers [PGWS19]. Hence, we can exclude managed software submission as an appropriate type for our platform as it violates *S1*, *S2*, and *S3*.

Potthast et al. [PGWS19] introduce the participant-in-charge software submission to address the problems of managed software submissions. This new approach enables participants to execute and edit their software without having to interact with the organizer. To do so, each participant receives a virtual machine with full privileges. They can write their source code and control the execution environment of the software inside the virtual machine. It is also possible to execute their software with available training data to find errors and to ensure their software's correctness. When they are sure that their software runs reliably, they can initiate an execution with the private data. For that, the platform transfers the virtual machines into a sandbox where the private data is available. However, the virtual machines have no Internet connection and will reset after the execution, and thus, only data in the provided output path are persistent. The results will only be published after an organizer review the output path to prevent data leaks [PGWS19].

We choose the participant-in-charge software submission for our platform as it fulfills all our requirements. It should be easy to add data packages to the data server, which provides data to the virtual machines. Additionally, the approach uses training data (i.e., public data) while testing and secured test data (i.e., private data) while evaluating. Both our data integrity requirements *D1* and *D2* are hereby fulfilled. Furthermore, new participants can add software easily (i.e., *S1*) by receiving a new virtual machine. Access to the virtual machines works via a provided login, and thus, the developed software is private for each participant (i.e., *S2*). As the participants have the login credentials and root privileges, they can alter their software at any time (i.e., *S3*) and perform test runs based on training data until they reach a satisfying level. All virtual machines gain the same resources (i.e., *S4*) to support fair competition.

Our concept is appropriate, as every member of the shared task is sufficiently covered. We cover the reduction of the evaluation process, introduction of a shared dataset, consideration of private and public data/software, and credibility of evaluation results. In the following, we present a more technical overview of the platform, based on TIRA [PGWS19].

5.1.2 Concept based on TIRA and Sampling Framework

Potthast et al. [PGWS19] introduce a platform named TIRA[1] Integrated Research Architecture (TIRA). The goal of TIRA is to tackle the large amount of uncoordinated shared tasks in computer science by providing a platform for organizers to host participant-in-charge software submissions. The new approach of handling software submission should motivate researchers to contribute to coordinated shared tasks and increase the overall reproducibility of the research area. We adapt our concept to support product sampling as a participant-in-charge software submission in TIRA. In the following, we present the overall architecture of TIRA, a sampling

[1] https://www.tira.io/

https://doi.org/10.24355/dbbs.084-202009211318-0

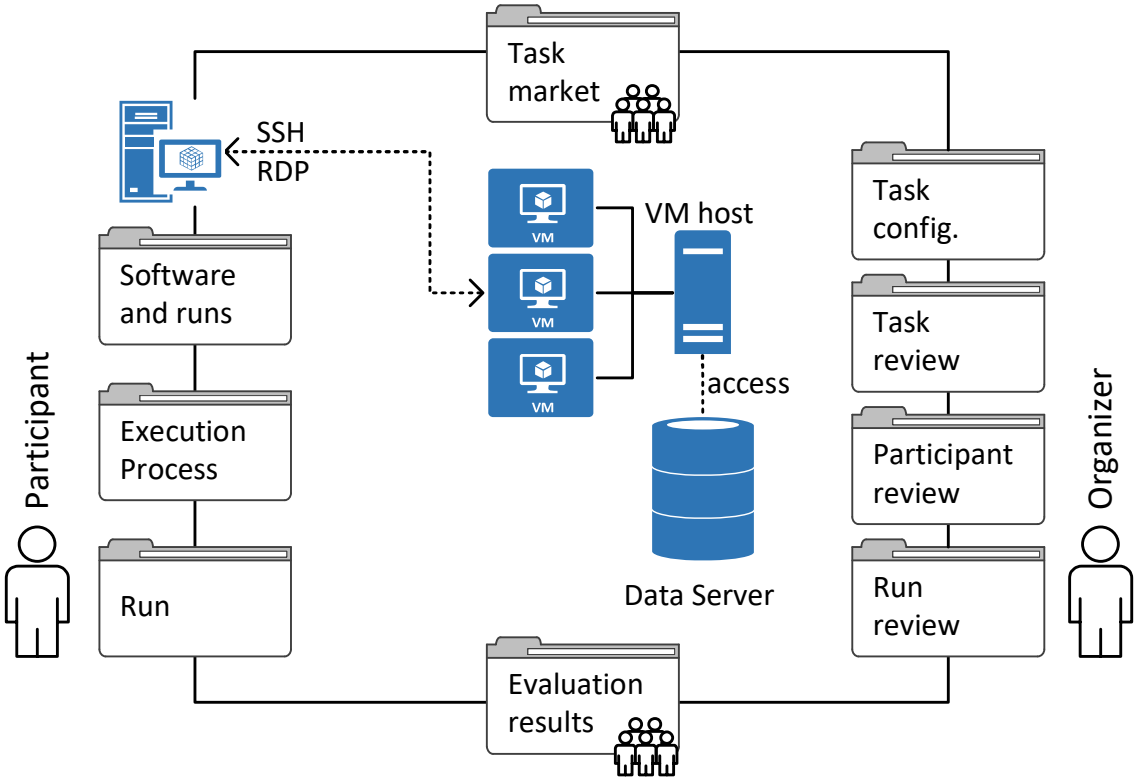


Figure 5.3: TIRA's interfaces for public (top, bottom), organizer (right), and participants (left). Figure designed by Potthast et al. [PGWS19] and adapted by us

framework that ensures fair competition between sampling algorithms, and the integration of our structure into TIRA.

TIRA's Architecture

TIRA supports a multitude of shared tasks, and when we speak of a task in context with TIRA, we mean a shared task. Figure 5.3 shows the interfaces for the public (top, bottom), the organizer (right), and the participants (left):

Organizer: Each organizer is responsible for the creation, configuration, and maintenance of a task. A sufficient description of the task and the data set is required to motivate other researchers to contribute. A submission of their algorithms requires a virtual machine for the participants. Distributing the machines and reviewing the participants are obligations of the organizer. The requirements for participant-in-charge software submission requires the organizer to review each run to prevent data leaks.

Public: The publicly available task market enables users to browse all tasks. Every user can read the task descriptions, ask for participation, and see the evaluation results. These results are shown in tables for each data package and contain an entry for each participant. Therefore, accessing results and comparing them is easy.

Participant: TIRA provides participants with a platform to experiment with their submission. For that, every participant receives a virtual machine which is accessible via SSH or RDP. The participant has root privileges on the virtual machine and can

https://doi.org/10.24355/dbbs.084-202009211318-0

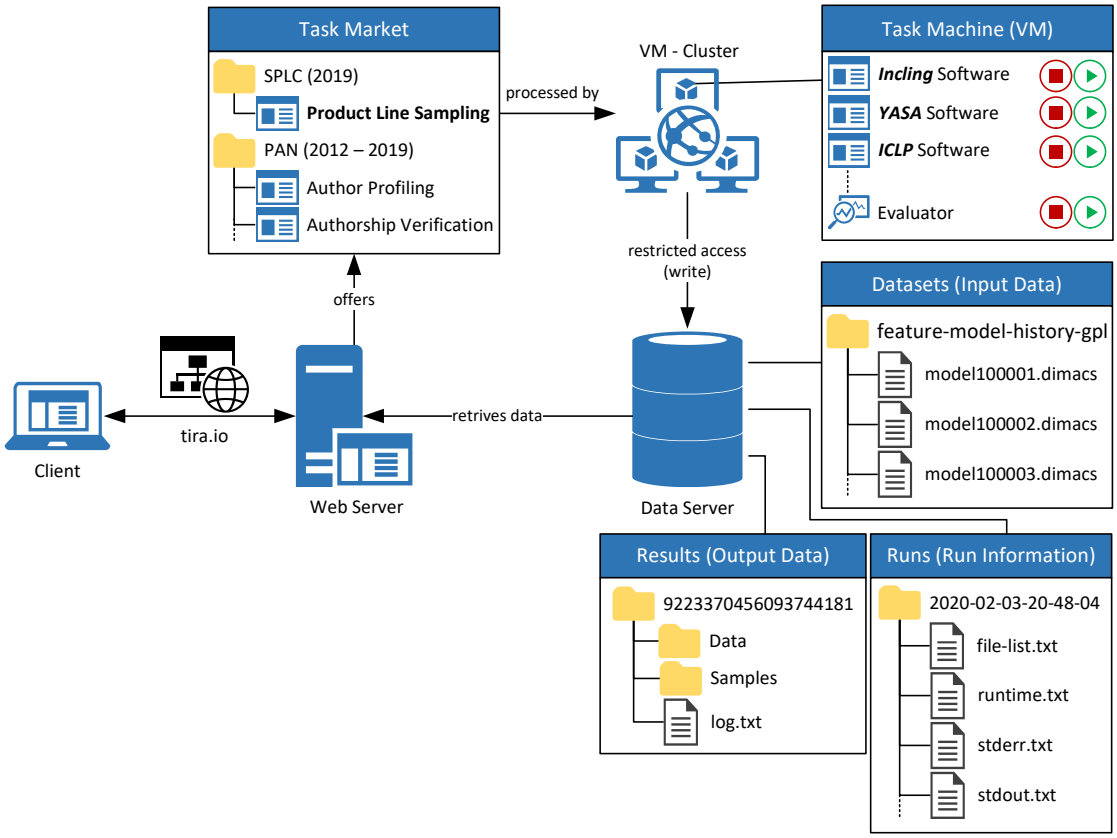


Figure 5.4: The general architecture of TIRA composed of the three main components: the webserver, the VM cluster, and the data server

freely set up his software and dependencies. Further, he can dictate the execution process of the software. The only requirement for the software is the ability to start its execution with a bash command. Starting the software with a bash command is possible via the user interface provided by TIRA.

In Figure 5.4, we show the general architecture of TIRA. TIRA advertises itself as *Evaluation as a Service*, a cloud-based service [PGWS19]. The architecture consists of three main components: a webserver, a data server, and a VM cluster. Accessing TIRA is possible via the website https://www.tira.io/ provided by the webserver. From there, one can browse the task market. Participants and organizers can log into their accounts and access their task machines. On these task machines, they can register, execute, and evaluate their software. For instance, a participant could set up the software for IncLing, YASA, and ICPL on one virtual machine.

However, only one software can be executed at a time. The last component is the data server that consists of three layers. TIRA stores the input data (e.g., the private and public data for our product sampling task) in the first layer. The second layer contains run information about the execution of a virtual machine. This information includes the console logs, a list of files changed by the execution, and system resource information, i.e., memory and CPU workload. The last layer stores the output data generated by a software run. In our case, we create a *Data* folder containing all results regarding the algorithm run's evaluation criteria.

https://doi.org/10.24355/dbbs.084-202009211318-0

Moreover, we save all samples generated by the evaluated algorithm as they are heavy to compute and could be useful for other use cases. For the final version of the platform, we remove the samples as they contain information about our dataset to prevent data leaks. Via the website, all users can access each task and their results. The webserver retrieves all vital information from the data server.

Framework for Sampling Challenge

Pett et al. [PTR+19] published a scalability challenge for product sampling [PTR+19]. They provide a dataset containing the history of three industrial large scale systems from different domains and motivate researchers to submit their computed samples and statistics. The authors used this information to compare the submission. To extend this challenge, we use TIRA to perform participant-in-charge submissions. However, constraints such as data formats, output formats, and profiling are still not addressed by TIRA. Questions such as: *What format should our algorithm support?*, *What output format should we support?*, *How are we sure other participants correctly measure their software?* are not answered and could prevent potential contributors to participate. For that, we propose a concept for a framework that profiles the software of the participants. To do so, we create a set of requirements for our framework to integrate into the sampling challenge as smooth as possible:

Requirements for the Sampling Framework:

- *F1*: The framework supports multiple input formats for feature models such as DIMACS [Cha93], FeatureIDE-XML [MTS+17], and GUIDSL [Bat05].
- *F2*: The submissions software support DIMACS as input format for feature models.
- *F3*: The framework supports only one output format for the computed sample (i.e., a data structure representing the sample).
- *F4*: The framework provides an interface to parse the results of sampling algorithms to the unified output format.
- *F5*: The framework is responsible to profile CPU time and memory of the executed software.
- *F6*: The framework proves the validity and achieved coverage of resulting samples to ensure credibility.
- *F6*: The framework calculates all evaluation criteria automatically.

In Figure 5.5, we present the concept and workflow for the product sampling framework. Generally, we differentiate between the framework and the participant domain (in green). The participant is responsible for providing software that computes samples and to implement an interface for the algorithm. In the following, we explain the general steps marked in the illustration:

1. We begin by reading all data from a given data path. This data consists of models that we evaluate in this run.

https://doi.org/10.24355/dbbs.084-202009211318-0

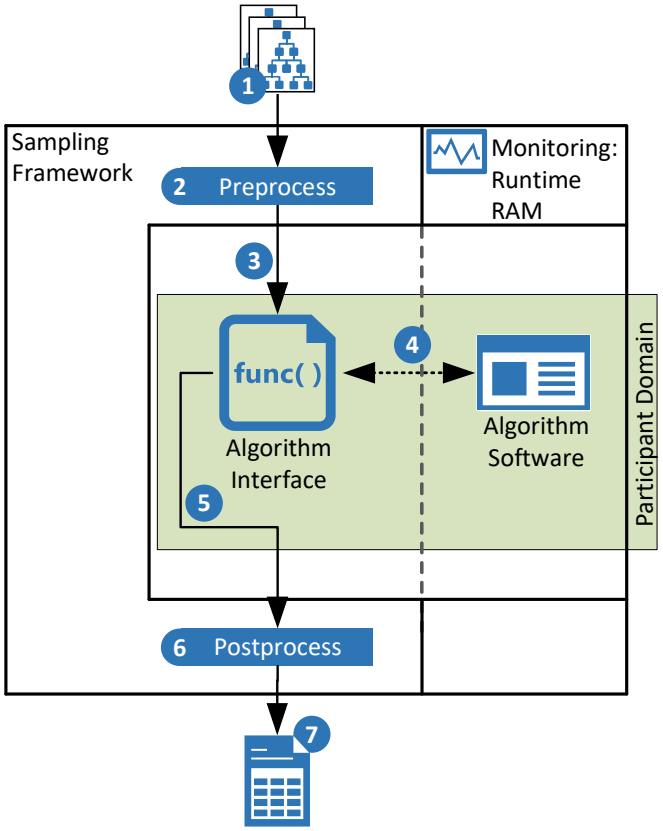


Figure 5.5: The general architecture and workflow for the sampling framework

2 The input models can have different formats (*F1*). Therefore, we can support a wide range of already available models. Individually deciding the format required by the participant's software is tedious. Therefore, we convert all models into DIMACS format (*F2*), a well known and efficient format to save feature models.

3 For each model, we execute the algorithm interface and pass over the model.

4 The interface is responsible for executing the participators software in a monitored environment. Inside the environment, we profile CPU runtime and memory usage (*F5*). When the software finishes its execution, then it informs the algorithm interface.

5 Rewriting already available implementations for sampling algorithms is tedious. Therefore, it is necessary to provide a transformation of the participators format into the format required by the framework(*F3 & F4*). This ability makes it easier to introduce software that is not primarily developed for the framework. After the output (i.e., the data structure describing the sample computed by the submission software) is parsed, it is then forwarded to the framework.

https://doi.org/10.24355/dbbs.084-202009211318-0

6. Now, we collect/calculate all information and bundle them:
Verification: At first, we use the output to verify each result by checking every sample for validity and calculating the achieved coverage (*F6*). This validation prevents cheating as modifications (e.g., intentionally reducing samples) are detected.
Extraction: Second, we extract sample size and sample runtime (*F7*) from the output and CPU profile data.
Memory: Third, we compute memory metrics (*F7*) from the memory profile data.
Sampling Similarity When the input data packages contain the evolution of a product line (i.e., feature models of the same product line retrieved at different timestamps), we compute the similarity (*F7*) between the current sample and the sample from the previous model.

7. Next, we save the information bundle in a data file in tabular form having the following data:

   - Identifier (Author, Algorithm)
   - Model Info (ID, Name, Number of (#) Features, #Constraints, #PossibleFeatureCombinations)
   - Sampling Info (Size, Runtime, Coverage, ROIC, MSOC, FIMDC)
   - Memory Info
   - Other (Timeout, Validity, T-value)

   When all models are processed, terminate, otherwise repeat from step three. An example table can be found in Figure A.1.

The presented framework fulfills all our requirements *F1* - *F7*. We paid much attention to the freedom of the participants. The interface supports any program and free execution of any software. The flexibility of the interface allows not only the usage of new software but also supports older software. The participants have no responsibility to measure CPU, memory, or evaluation criteria. Having the same dataset and the same procedure to profile and evaluate the results of different participants makes the result easy to compare.

Integration of Sampling Framework and TIRA

We use TIRA for handling the highly flexible participant-in-charge software submissions. Additionally, we provide a framework to manage the automated process of sampling algorithms and their comparison. However, the effort to set up a framework could scare participants and prevent their submission. Therefore, we integrate our framework in TIRA. We install and configure the framework on every virtual machine before their distribution. The participants need to implement the interface of the framework and store their implementation in a given path. Via TIRA, it is possible to execute our platform, which indirectly executes the submitted software provided by the participant.

https://doi.org/10.24355/dbbs.084-202009211318-0

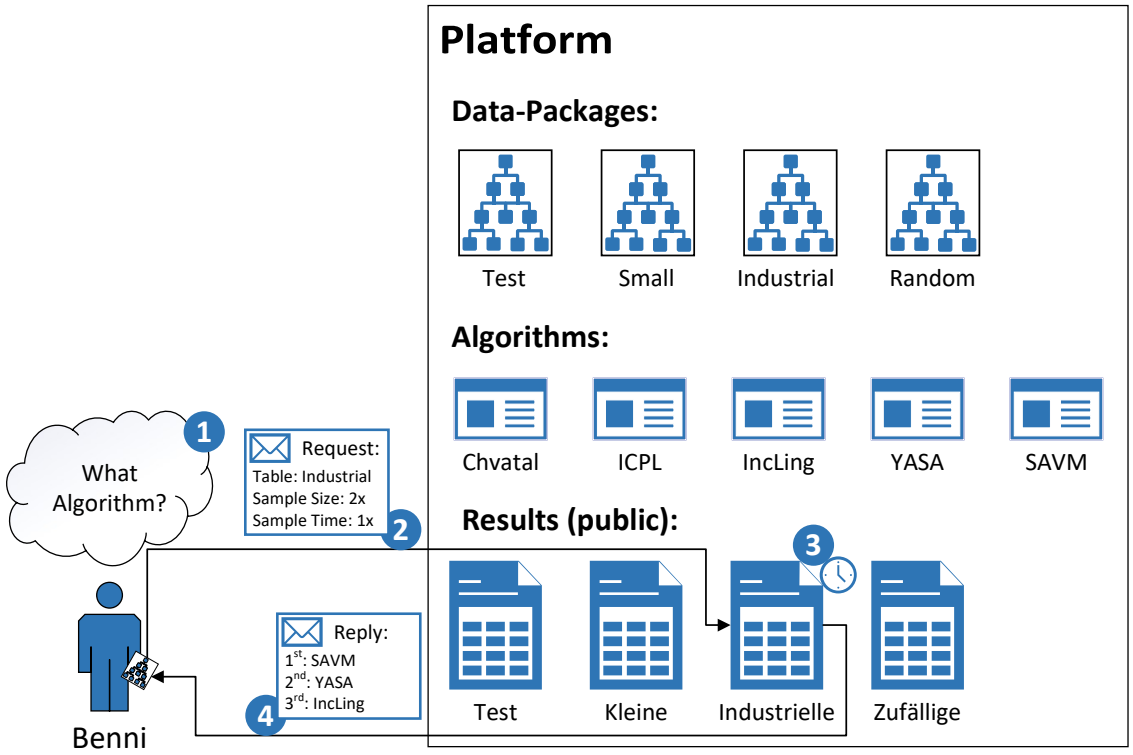


Figure 5.6: The general concept for the requirements evaluator addressing the problem for industrial users to select an appropriate sampling algorithm

5.2 Priority-based Selection of Appropriate Algorithms

In this section, we use our platform's results to compute recommendations of sampling algorithms for industrial users. Determining an appropriate algorithm for t-wise sampling is hard as many algorithms are available, and each algorithm fulfills evaluation criteria to a different degree. This chaos makes it hard for industrial users to select an appropriate algorithm that also achieves their objectives (e.g., small sample sizes) and constraints (e.g., only a limited amount of RAM available). Based on the data generated by our platform, we propose a concept for an evaluator that considers the requirements of the user.

5.2.1 Requirement Evaluator

Using product lines in the industry is a common approach to handle the variability of systems, and testing them is an essential process [BRN+13, MNM+18]. However, industrial users are often not familiar with the current state of the research, and, thus, selecting an appropriate algorithm is hard. The most important aspect is to maximize the return value for the user. To determine one or two sampling algorithms out of more than 38 available sampling algorithms is complicated, especially as the comparison between them is lacking [VAHT+18]. With the introduction of our platform in the research area, we aim to fill up the missing comparisons between sampling algorithms. Further, we aim to provide an evaluator that calculates a recommendation of an appropriate sampling algorithm while considering the user's priorities in terms of evaluation criteria.

https://doi.org/10.24355/dbbs.084-202009211318-0

In Figure 5.6, we show the general process of such an evaluator. Our industrial user, Benni, wants to select a sampling algorithm for his product line. Traditionally, Benni could select an industrial solution that computes the sample while having no information about the used algorithm and no room for improvements. Otherwise, he incorporates into the research area and spends much effort to determine the best algorithm by performing several evaluations on his product line. In short, Benni could decide for a non-optimal solution with less effort (i.e., the industrial approach) or an optimal solution with enormous effort (i.e., manually comparing algorithms). Our approach aims to provide Benni an optimal solution with less effort. The process is as followed:

1. Benni needs to select a sampling algorithm for his newest product line. He decides to use our platform.
2. Benni knows that their product line is large and represents a large scale system, and thus, he decides to request the result table for the industrial-sized data. He knows that testing a single product of the product line is complex and that he requires a sampling algorithm that focuses more on sampling size. Hence, Benni makes a request that weights the sampling size twice as much as the sampling time.
3. The platform receives the request and starts the requirements evaluator on the selected results table with the given emphasis.
4. The platform sends a reply to Benni after the evaluator finishes. The reply contains a sorted list of sampling algorithms that we recommend. For instance, Benni receives the recommendation to use SAVM, followed by YASA, and IncLing as alternatives.

In Figure 5.7, we show the internal process of the platform and the evaluator. Each colored entry at the top of the figure represents one entry of the results table. The platform provides the evaluator with these results. Afterward, the evaluator constructs a recommendation table based on the given results by computing a heuristic for each entry. We name the heuristic for a sample its' *score*. We propose various approaches for the calculation of the score, which have different score policies. Approaches that enforce the lowest-best policy compute a small score for algorithms that perform well. On the contrary, an approach with the highest-best policy computes a high score for useful algorithms. Because of that, we decided to introduce a *rank* to state the order of our recommendations clearly. The recommendation table contains the following entries:

Rank a numerical index showing the ranking of the scores. The lowest rank is the most recommended algorithm.

Score is a numerical value showing how good or bad an algorithm performs for the given requirements based on the platform's data.

[2x] Size the numerical value of the score contributed by given evaluation criteria. For our example, the numerical value contributed by the double-weighted sampling size. For every evaluation criterion, we add a new table column, showing the weight of the criteria in the column title (e.g., [1x] Runtime).

https://doi.org/10.24355/dbbs.084-202009211318-0

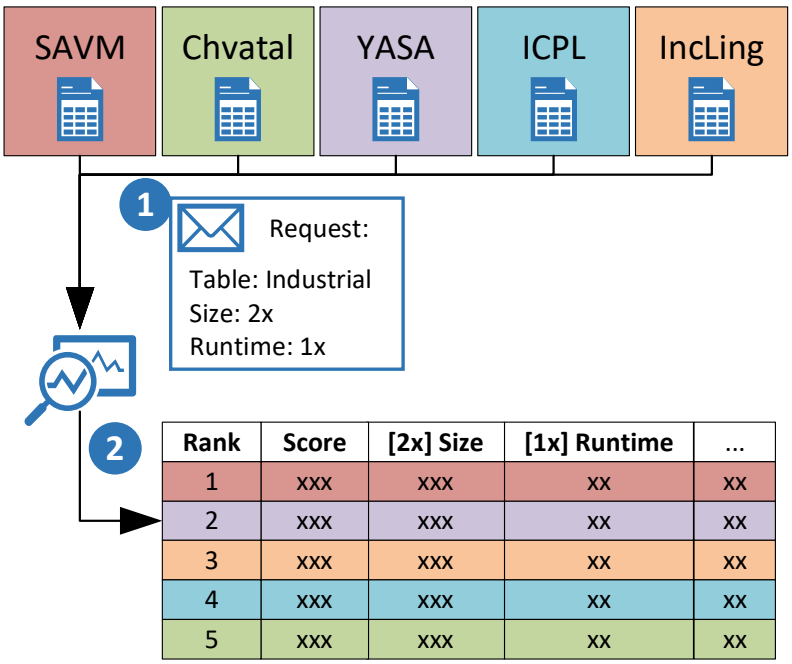


Figure 5.7: The process to select the most appropriate algorithm. The results for each algorithm (top) are forwarded to the requirement evaluator. Afterward, the evaluator computes a score for every input and generates a recommendation table (bottom) showing the most appropriate algorithm with the lowest rank

5.2.2 Computation of Scores

An entry that we evaluate in the requirement evaluator is called a *run* from now on. The most straightforward approach to recommend an algorithm would be to select the algorithm that performs best for a given criterion. However, the algorithm that performs best for one criterion could perform extremely bad for other criteria. Therefore, computing an appropriate score needs to consider not only one evaluation criterion but all. Our approaches calculating the score of a run considers the following criteria: sampling size, sampling time, achieved coverage, sampling similarity, and memory consumption. Additionally, the user can provide a prioritization of the criteria, and therefore, compute a recommendation that best suits his needs.

Example 5.1: Score Prioritization

A user wants to compute an appropriate algorithm for his product line. However, he is not interested in the similarity of samples and more in an efficient runtime. Hence, he could use our evaluator with the following prioritization:

Criterion	Weight (Priority)
Size	1
Time	2
Coverage	1
Similarity	0
Memory	0

Abbreviated form:
[S:1, T:2, C:1, Sim:0, M:0]

https://doi.org/10.24355/dbbs.084-202009211318-0

A prioritization of zero in sampling similarity does not mean that the resulting algorithm is terrible at computing similar samples. It just ignores the results of the similarity criterion when computing the score.

Definition 5.2: Score Prioritization

A score prioritization Ω *is given by a 5-tuple* $\Omega = (\omega_s, \omega_t, \omega_c, \omega_{sim}, \omega_m)$ *where:*

- $\omega_s \in \mathbb{R}$ *is the weight for the sample size*
- $\omega_t \in \mathbb{R}$ *is the weight for the sample time*
- $\omega_c \in \mathbb{R}$ *is the weight for the sample coverage*
- $\omega_{sim} \in \mathbb{R}$ *is the weight for the sample similarity*
- $\omega_m \in \mathbb{R}$ *is the weight for the sample memory*

We introduce two approaches to compute the score. Our first approach computes nominal values (i.e., values between 0 and 1) for every criterion and combines them to create the score. The second approach combines the individual rankings of the runs related to one evaluation criterion (i.e., the ranking of sampling sizes, ranking coverage). For the second approach, we developed a simple and two optimized variants.

The input for our approaches is always the data file generated by the sampling framework (cf. example in Figure A.1). As the framework generates a file entry for each model in the data package, it is necessary to reduce all entries into one. To do so, we compute averages for the following entries: Sample Size, Sample Time, Sample Coverage, Sample Similarity (FIMDC), and Memory Consumption. Further, we compute averages of nominal values for our nominal-based approach so that a high value indicates a *good* sampling process:

Size: Product sampling aims to compute configurations that cover as many feature interactions as possible, and thus, normalizing the size of samples to all possible valid feature interactions is possible. For that, we compute a quotient by dividing the sample size with all possible interactions. The resulting quotient explains the number of configurations required for each interaction. A high quotient indicates a worse sample process, and thus, we invert the quotient by subtracting it from one (i.e., $1 -$ quotient).

Runtime: The framework sets a timeout for every performed run. We can use the timeout to compute the nominal value by dividing the runtime with the timeout. A high quotient indicates that more time was needed, and thus, we need to invert the quotient by subtracting it from one (i.e., $1 -$ quotient).

T-Wise Coverage: Sampling algorithms aim to compute samples for a given t-value. It is essential to cover as many (preferably all) t-wise feature interactions. The coverage indicates the percentage of these interactions included in the sample. Our framework computes the coverage as a value between 0 and 1 (i.e., 0 means 0%,

https://doi.org/10.24355/dbbs.084-202009211318-0

while 1 means 100% of the feature interactions were covered). Hence, the computed coverage is already normalized and can be used directly for NBS. Furthermore, high coverage values show that an algorithm is better than one with lower coverage, so it is in consent with our highest-best policy.

Similarity: Feature models tend to evolve over time [TBK09]. Evolutions such as adding new features, removing old features, or changing the hierarchy of existing features can happen. After every change, we need to compute a new sample for testing. Now, it is possible to compute the similarity of the sample before and after a change. Pett et al. [Pet18a] introduce several metrics for computing the similarity between two samples. FIMDC is the metric that shows the best results, and thus, we use FIMDC as the indicator for sampling similarity. The calculation results in a value between 0 (i.e., samples are not similar at all) and 1 (i.e., samples are identical), and thus, we can use FIMDC directly as it is already normalized. The similarity value of two samples may support testing, depending on the testing strategy selected by the user [HB10, Pet18a]. There are two use cases for similarity. First, the user wants to find bugs in system functionalities they have already seen. Thus, a high similarity value is good because our sample after an evolution is very similar to the previous one. Second, the user wants to find bugs in system functionalities that he has not seen before. Thus, a low similarity value is good because more different configurations are tested compared to the previous sample. The use case depends on the user. So for our approaches, we decided to let the user control the use case via the given score prioritization. A positive weight for sampling stability ($\omega_{sim} \in \mathbb{R}^+$) favors a high similarity, while a negative value ($\omega_{sim} \in \mathbb{R}^-$) favors a low similarity value.

Memory: The memory consumption measured by the framework shows how much memory was created and freed by the sampling algorithm. As we have no upper boundary given, it is hard to normalize it, and we would typically drop the memory aspect when calculating the nominal-based approach. However, we normalize the spend memory by dividing it with the number of all possible valid feature interactions for testing purposes. It should have no negative impact on the score as we can always set the memory prioritization to zero to prevent distorted results.

We construct the data structure `Framework-Data` over the computed averages:

Definition 5.3: Framework-Data

A framework data D is given by a 8-tuple $D = (s, t, c, sim, m, n_s, n_t, n_m)$ where:

- $s \in \mathbb{R}$ *is the average sample size*
- $t \in \mathbb{R}$ *is the average sample time*
- $c \in \mathbb{R}$ *is the average sample coverage*
- $sim \in \mathbb{R}$ *is the average sample similarity*
- $m \in \mathbb{R}$ *is the average sample memory*
- $s^N \in \mathbb{R}$ *is the normalized value for sample size*
- $t^N \in \mathbb{R}$ *is the normalized value for sample time*

https://doi.org/10.24355/dbbs.084-202009211318-0

- $m^N \in \mathbb{R}$ *is the normalized value for sample memory*

We define $\mathbb{SD}$ *as the superset of all framework data. For the following definitions, we introduce some notations:*

1. $v_D = v \in D$ *for accessing the element* v $(s, t, c, sim, s^N, t^N, m^N)$ *from* D.
2. *A function defined as* $f^x(v) : \mathbb{N} \to \mathbb{N}$ *with* $f^x(v) = x * v$ *and* $x = \{s, t\}$ *can be applied to both* s *and* t. *For instance, we could use the definition above as* $f^s(v) = s * v$ *or as* $f^t(v) = t * v$.

In the following subsections, we introduce the different approaches and define them formally. Afterward, we show a detailed example of the computation for all approaches based on the same data.

5.2.2.1 Nominal-Based Score (NBS) (highest-best)

The idea of NBS is to compute subscores for each criterion. For that, we introduce a mapping for each criterion to a nominal value. All necessary values are already normalized and present in the `Framework-Data`. With that, we regard each normalized values multiplied by their prioritization as a subscore. Then we compute NBS as the sum of all subscores:

Definition 5.4: Nominal-Based Score

Let $D \in \mathbb{SD}$ *be a given framework data. Let* $\Omega = (\omega_s, \omega_t, \omega_c, \omega_{sim}, \omega_m)$ *be a given score prioritization. Then, we can compute the nominal-based score* S_{NBS} *based on the sum of nominal values for all criteria.* $S_{NBS} : \mathbb{SD} \to \mathbb{R}$ *with*

$$S_{NBS}(D) = \left(\omega_s * s_D^N\right) + \left(\omega_t * t_D^N\right) + \left(\omega_c * c_D\right) + \left(\omega_{sim} * sim_D\right) + \left(\omega_m * m_D^N\right) \quad (5.1)$$

NBS has multiple advantages, such as extensibility and independency. For example, extending the score computation is easy as multiplying/adding a new subscore is possible. Further, independence, in this case, describes the ability to calculate the score without depending on results from other algorithms.

Nonetheless, NBS also has a weakness. There is no defined relationship between the multiple criteria, and thus, computing a score that treats all subscores equally is hard. This undefined relationship leads to the fact that some subscores have more impact on the score than other subscores. For instance, the impact of the sampling size subscore is high as an increase in the size has almost no impact on the quotient (i.e., the number of feature interactions is significantly higher than the sampling size). Furthermore, our sampling quotient will never reach a low value as the sampling process aims to compute small samples.

On the contrary, the runtime score growths continuously with a much higher chance to reach his maximum value (i.e., the timeout), and thus, has less impact on the

https://doi.org/10.24355/dbbs.084-202009211318-0

score (i.e., a higher runtime lead to a smaller quotient which leads to a smaller impact for the subscore). This weakness gets more distinctive when considering a prioritization such as [S:5, T:5, C:1, Sim:1]. The runtime generally has a lower impact than the sampling size. When both criteria are equally scaled, then the lower impact of the runtime is also scaled accordingly. This lower impact could lead to a recommendation that does not favor both size and runtime but only algorithms that compute small samples. The reason is that NBS is a highest-best score, and the impact of the size score is high, and therefore, the score growths high. All in all is our given formal definition sufficient for NBS.

5.2.2.2 Simple Rank-Based Score (SRBS) (lowest-best)

Instead of individually computing the score with nominal values, it is possible to compute the score for a set of runs. Our second approach uses the rankings for each criterion as subscores and aggregates the final score. A low rank indicates a more efficient sampling performance, and thus, the simple ranking-based score (SRBS) is a lowest-best score. We begin by sorting the sampling size and time in ascending order (i.e., a low value is good) and coverage and similarity in descending order (i.e., a high value is good). Then, we determine the ranking for each criterion by assigning a rank to every run. The ranking starts with the integer one and is increased by one integer for every run. We use the ranking for every criterion as the subscore and combine them to form the score for each run. We define SRBS formally as followed:

Definition 5.5: Simple Rank-Based Score

Let $\Psi = \{D_i \in \mathbb{SD} \mid i \in \{0, \ldots, n\}\}$ *with* $n \in \mathbb{N}$ *be a given set of framework data. Let* $\Omega = (\omega_s, \omega_t, \omega_c, \omega_{sim}, \omega_m)$ *be a given score prioritization. We can construct sets over* Ψ *that contain all specific elements of all framework data with* $x = \{s, c, t, sim, m\}$*:*

$$\Psi_x = \bigcup_{0}^{n} x_{D_i} \tag{5.2}$$

The ranking function determines the rank of a given element inside their set. We compute the rank of a data's element x *depending on how many elements besides* x *have a better value. We have both values that are good whenever they are low, and whenever they are high, so we need to introduce two ranking functions. The function* $r^<$ *determines the rank when low values are better while* $r^>$ *determines the rank when high values are better.*

$$r_x^< : \underbrace{\mathbb{SD} \times \cdots \times \mathbb{SD}}_{n\text{-}times} \times \mathbb{SD} \to \mathbb{N} \text{ with } r_x^<(\Psi, D) = |\{v_i \in \Psi_x \mid v_i \leq x_D\}| \tag{5.3}$$

$$r_x^> : \mathbb{SD} \times \cdots \times \mathbb{SD} \times \mathbb{SD} \to \mathbb{N} \text{ with } r_x^>(\Psi, D) = |\{v_i \in \Psi_x \mid v_i \geq x_D\}| \tag{5.4}$$

https://doi.org/10.24355/dbbs.084-202009211318-0

With the ranking functions, it is possible to calculate the simple rank-based score $S_{SRBS} : \mathbb{SD}_0 \times \cdots \times \mathbb{SD}_n \times \mathbb{SD} \to \mathbb{R}$ *with*

$$\begin{aligned} S_{SRBS}(\Psi, D) = &\big(\omega_s * r_s^<(\Psi, D)\big) + \big(\omega_t * r_t^<(\Psi, D)\big) + \\ &\big(\omega_c * r_c^>(\Psi, D)\big) + \big(\omega_{sim} * r_{sim}^>(\Psi, D)\big) + \\ &\big(\omega_m * r_m^>(\Psi, D)\big) \end{aligned} \tag{5.5}$$

The SRBS is expandable and considers the individual criteria separately. We can ensure fair treatment of the individual criteria by calculating the same value range for each criterion as a partial score. Therefore each criterion has the same influence on the overall score. When a user wants to weight one criterion more than the others, it is possible to change the prioritization accordingly.

However, SRBS also has several disadvantages. One of them is that we always need the list of all runs to calculate the SRBS. Furthermore, by assigning integer ascending ranks, the results within each criterion are attenuated. For example, let us look at the algorithms A, B, and C. Let us assume we have the following averages of sample sizes given A=33, B=31, and C=150. SRBS would produce the following subscores (ranks) A=2, B=1, and C=3. Therefore, C's bad performance only impacts the overall score by three, while A has an impact of two. However, their sample sizes are incredibly different. This example shows that SRBS treats everything equally, criteria, and runs. We decided to improve the SRBS to calculate the score equally for the individual criteria while considering the gaps between multiple runs more heavily. We named the resulting variant the weighted rank-based score.

5.2.2.3 Weighted Rank-Based Score (WRBS) (lowest best)

The weighted rank-based score is an extension of SRBS. Our intention is to counteract the problem that criteria are not treated fairly. Like the SRBS, the WRBS is a lowest-best score.

The calculation of the WRBS differs from the SRBS only by the assignment of the ranks. Instead of counting up ranks from one, we determine the ranks relative to the run with the best result. The rank of a run is calculated by dividing its value by the value of the best run. Therefore the best run always has the rank one while other runs have higher ranks. To retake the previous example, we assume the following sampling sizes: A=33, B=31, and C=150. With WRBS, the following ranks would result in A=1.065, B=1, C=4.839. From the results, we can see that C gets a generally higher score (i.e., worse result) than A and B.

Since we no longer distribute integer ranks but consider the ranks as a kind of weighting of the best result, we name the approach the weighted rank-based score (WRBS). We formally define WRBS as follows:

Definition 5.6: Weighted Rank-Based Score

Let $\Psi = \{D_i \in \mathbb{SD} \mid i \in \{0, \ldots, n\}\}$ with $n \in \mathbb{N}$ be a given set of framework data. Let $\Omega = (\omega_s, \omega_t, \omega_c, \omega_{sim}, \omega_m)$ be a given score prioritization. Let Ψ_x (cf. Equation 5.2) describe the construction of sets that contain all values of a

https://doi.org/10.24355/dbbs.084-202009211318-0

specific element for all D_i. We define w^{min} as the function that calculates the weighted rank for elements where a low value indicate better performance (e.g., sampling size and runtime):

$$w_x^{min} : \underbrace{\mathbb{SD} \times \cdots \times \mathbb{SD}}_{\text{n-times}} \times \mathbb{SD} \rightarrow \mathbb{R} \text{ with } w_x^{min}(\Psi, D) = \frac{x_D}{min(\Psi_x)} \tag{5.6}$$

We define w^{-max} as the function that calculates the weighted rank for elements where a high value indicate better performance (e.g., coverage and similarity). As these values are normalized and range between [0,1], we further need to inverse the quotient to produce a low value for an excellent performance.

$$w_x^{-max} : \underbrace{\mathbb{SD} \times \cdots \times \mathbb{SD}}_{\text{n-times}} \times \mathbb{SD} \rightarrow \mathbb{R} \text{ with } w_x^{-max}(\Psi, D) = \frac{max(\Psi_x)}{x_D} \tag{5.7}$$

With the ranking functions, it is possible to calculate the weighted rank-based score $S_{WRBC} : \mathbb{SD}_0 \times \cdots \times \mathbb{SD}_n \times \mathbb{SD} \rightarrow \mathbb{R}$ for $D \in \Psi$ with

$$\begin{aligned} S_{WRBC}(\Psi, D) = &\big(\omega_S * w_s^{min}(\Psi, D)\big) + \big(\omega_T * w_t^{min}(\Psi, D)\big) + \\ &\big(\omega_C * w_c^{-max}(\Psi, D)\big) + \big(\omega_{Sim} * w_{sim}^{-max}(\Psi, D)\big) + \\ &\big(\omega_m * w_m^{min}(\Psi, D)\big) \end{aligned} \tag{5.8}$$

WRBS has the same advantages as SRBS and solves the disadvantage of unfair rank assignments. By weighting the ranks, we achieve a much more precise differentiation of the individual runs. Therefore, it is more likely that bad algorithms will get a worse score and that two algorithms with similar values will also get a similar score. Most people associate a high score with better performance, and hence, we introduce another variant of the WRBS that computes a high score for good performances.

5.2.2.4 Inverse Weighted Rank-Based Score (IWRBS) (highest-best)

The inverse weighted rank-based score (IWRBS) extends the WRBS by inverting the rank assignments. Therefore, it is a highest-best score. We define IWRBS formally as follows:

Definition 5.7: Inverse Weighted Rank-Based Score

Let $\Psi = \{D_i \in \mathbb{SD} \mid i \in \{0, \ldots, n\}\}$ with $n \in \mathbb{N}$ be a given set of framework data. Let $\Omega = (\omega_s, \omega_t, \omega_c, \omega_{sim}, \omega_m)$ be a given score prioritization. Let Ψ_x (cf. Equation 5.2) describe the construction of sets that contain all values of a specific element for all D_i. Let w_x^{min} (cf. Equation 5.6) and w_x^{-max} (cf. Equation 5.7) be the ranking functions to determine the rank of a given data

https://doi.org/10.24355/dbbs.084-202009211318-0

framework. With the ranking functions, it is possible to calculate the inverted weighted rank-based score $S_{IWRBC} : \mathbb{SD}_0 \times \cdots \times \mathbb{SD}_n \times \mathbb{SD} \to \mathbb{R}$ for $D \in \Psi$ with

$$\begin{aligned} S_{IWRBC}(\Psi, D) = &\big(\omega_s * w_s^{-max}(\Psi, D)\big) + \big(\omega_t * w_t^{-max}(\Psi, D)\big) + \\ &\big(\omega_c * w_c^{min}(\Psi, D)\big) + \big(\omega_{sim} * w_{sim}^{min}(\Psi, D)\big) + \\ &\big(\omega_m * w_m^{-max}(\Psi, D)\big) \end{aligned} \tag{5.9}$$

The IWRBS delivers not entirely inverted results of the WRBS. We do not directly invert the WRBS, and therefore, both scores do not always recommend the same algorithm. The reason for that is when we use WRBS, we assign the best run as rank one. The ranks for other runs are weights relative to the best value. This weighting leads to almost the same results for algorithms that are close to the best one.

On the contrary, it also leads to awful scores for outliers. Now, when we invert WRBS, then all runs are weighted based on the worst value. This inversion could lead to higher gaps between algorithms with similar good values, and therefore, produce different recommendations. This difference is also why we handle IWRBS as an individual variant instead of the inversion of WRBS.

5.2.3 Computing Recommendations

We defined our approaches for the requirement evaluator. Now, we show how to calculate all approaches and extract a recommendation. For our example, we computed and modified the framework data with t=2 for the fictional algorithms A, B, C, D. In Table 5.1, we collect the averages and necessary information in the form of framework data (cf. Definition 5.3).

NBS: Assume A be the given framework data for algorithm A (cf. row A in Table 5.1). Assume $\Omega = (1, 1, 1, 1, 1)$ be the given score prioritization. We can compute $S^+_{NBS}(A)$ as described in Equation 5.1:

$$\begin{aligned} S_{NBS}(A) &= \big(\omega_s * s_A^N\big) + \big(\omega_t * t_A^N\big) + \big(\omega_c * c_A\big) + \big(\omega_{sim} * sim_A\big) + \big(\omega_m * m_A^N\big) \\ &= \big(1 * 0.99994\big) + \big(1 * 0.9654\big) + \big(1 * 1\big) + \big(1 * 0.752\big) + \big(1 * 0.9099\big) \\ &= 0.99994 + 0.9654 + 1 + 0.752 + 0.9099 \\ &= \underline{\underline{4.627}} \end{aligned}$$

Algorithm	**s**	**t in ms**	**c**	**sim**	**m in MB**	$\mathbf{s^N}$	$\mathbf{t^N}$	$\mathbf{m^N}$
A	36	110698	1	0.752	54414	0.99994	0.9654	0.9099
B	37	6262	1	0.742	2743	0.99993	0.9979	0.9953
C	69	2977	1	0.572	499	0.99988	0.9992	0.9994
D	42	3137	0.95	0.566	499	0.99993	0.9991	0.9999

Table 5.1: The sampling framework generates data for all algorithms. We generated all entries with our sampling framework. We have modified the data to show some differences between the different approaches better. Our modification does not represent any actual results, and their sorely purpose is to show the detailed calculation for each approach

https://doi.org/10.24355/dbbs.084-202009211318-0

The calculations for B, C, and D are analogous and produce the following NBSs (B=4.735, C=4.570, D=4.515). We calculate all scores before starting to recommend algorithms.

SRBS: Assume $\Psi = (A, B, C, D)$ be the given set of framework data for the algorithms A, B, C, and D (cf. row A-D in Table 5.1). Assume $\Omega = (1, 1, 1, 1, 1)$ be the given score prioritization. We begin by constructing the filter sets for each criterion and use the ranking functions $r_x^<$ and $r_x^>$ to determine the rank for data set A in each filter set:

$$\begin{array}{lll}
r_s^<(\Psi, A) & \text{with } \Psi_s & = \{36, 37, 69, 42\} \rightarrow 1 \\
r_t^<(\Psi, A) & \text{with } \Psi_t & = \{110698, 6262, 2977, 3137\} \rightarrow 4 \\
r_c^>(\Psi, A) & \text{with } \Psi_c & = \{1, 0.95\} \rightarrow 1 \\
r_{sim}^>(\Psi, A) & \text{with } \Psi_{sim} & = \{0.752, 0.742, 0.572, 0.566\} \rightarrow 1 \\
r_m^>(\Psi, A) & \text{with } \Psi_m & = \{54414, 2743, 499\} \rightarrow 3
\end{array}$$

Then, we can compute the $S_{SRBS}(\Psi, A)$ for A as described in Equation 5.5:

$$\begin{aligned}
S_{SRBS}(\Psi, A) = & \big(\omega_s * r_s^<(\Psi, D)\big) + \big(\omega_t * r_t^<(\Psi, D)\big) + \\
& \big(\omega_c * r_c^>(\Psi, D)\big) + \big(\omega_{sim} * r_{sim}^>(\Psi, D)\big) + \\
& \big(\omega_m * r_m^<(\Psi, D)\big) \\
= & \big(1 * r_s^<(\Psi, D)\big) + \big(1 * r_t^<(\Psi, D)\big) + \\
& \big(1 * r_c^>(\Psi, D)\big) + \big(1 * r_{sim}^>(\Psi, D)\big) + \\
& \big(1 * r_m^<(\Psi, D)\big) \\
= & \big(1 * 1\big) + \big(1 * 4\big) + \big(1 * 1\big) + \big(1 * 1\big) + \big(1 * 3\big) \\
= & \underline{\underline{10}}
\end{aligned}$$

The calculations for B, C, and D are analogous and produce the following SRBSs (B=10, C=10, D=12).

WRBS: Assume that $\Psi = (A, B, C, D)$ is the given set of framework data for the algorithms A, B, C, and D (cf. row A-D in Table 5.1). Assume that $\Omega = (1, 1, 1, 1, 1)$ is the given score prioritization. Assume that Ψ_x are the filtered sets constructed in the SRBS example. We begin by using the ranking functions w_x^{min} and w_x^{-max} to determine the rank for data set A for each criterion:

https://doi.org/10.24355/dbbs.084-202009211318-0

$$
\begin{aligned}
w_s^{min}(\Psi, A) &= \frac{s_A}{min(\Psi_s)} &&= \frac{36}{36} &&= 1 \\
w_t^{min}(\Psi, A) &= \frac{t_A}{min(\Psi_t)} &&= \frac{110698}{2977} &&= 37.18 \\
w_c^{-max}(\Psi, A) &= \frac{max(\Psi_c)}{c_A} &&= \frac{1}{1} &&= 1 \\
w_{sim}^{-max}(\Psi, A) &= \frac{max(\Psi_{sim})}{sim_A} &&= \frac{0.75}{0.75} &&= 1 \\
w_m^{min}(\Psi, A) &= \frac{m_A}{min(\Psi_m)} &&= \frac{54414}{499} &&= 109.05
\end{aligned}
$$

Then, we can compute the $S_{WRBS}(\Psi, A)$ for A as described in Equation 5.8:

$$
\begin{aligned}
S_{WRBS}(\Psi, A) = &\big(\omega_s * w_s^{min}(\Psi, D)\big) + \big(\omega_t * w_t^{min}(\Psi, D)\big) + \\
&\big(\omega_c * w_c^{-max}(\Psi, D)\big) + \big(\omega_{sim} * w_{sim}^{-max}(\Psi, D)\big) + \\
&\big(\omega_m * w_m^{min}(\Psi, D)\big) + \\
= &\big(1 * w_s^{min}(\Psi, D)\big) + \big(1 * w_t^{min}(\Psi, D)\big) + \\
&\big(1 * w_c^{-max}(\Psi, D)\big) + \big(1 * w_{sim}^{-max}(\Psi, D)\big) + \\
&\big(1 * w_m^{min}(\Psi, D)\big) + \\
= &\big(1 * 1\big) + \big(1 * 37.18\big) + \big(1 * 1\big) + \big(1 * 1\big) + \big(1 * 109.05\big) \\
= &\underline{\underline{149.23}}
\end{aligned}
$$

The calculation for B, C, and D is analogous and produce the following WRBSs (B=10.64, C=6.23, D=5.60).

IWRBS: Assume that $\Psi = (A, B, C, D)$ is the given set of framework data for the algorithms A, B, C, and D (cf. row A-D in Table 5.1). Assume that $\Omega = (1, 1, 1, 1, 1)$ is the given score prioritization. Assume that Ψ_x are the filtered sets constructed in the SRBS example. We begin by using the ranking functions w_x^{max} and w_x^{-min} to determine the rank for data set A for each criterion:

$$
\begin{aligned}
w_s^{-max}(\Psi, A) &= \frac{max(\Psi_s)}{s_A} &&= \frac{69}{36} &&= 1.917 \\
w_t^{-max}(\Psi, A) &= \frac{max(\Psi_t)}{t_A} &&= \frac{110698}{110698} &&= 1 \\
w_c^{min}(\Psi, A) &= \frac{c_A}{min(\Psi_c)} &&= \frac{1}{0.95} &&= 1.053 \\
w_{sim}^{min}(\Psi, A) &= \frac{sim_A}{min(\Psi_{sim})} &&= \frac{0.75}{0.56} &&= 1.329 \\
w_m^{-max}(\Psi, A) &= \frac{max(\Psi_m)}{s_A} &&= \frac{54414}{54414} &&= 1
\end{aligned}
$$

https://doi.org/10.24355/dbbs.084-202009211318-0

Algorithm	NBS	SRBS	WRBS	IWRBS
A	4.627	10	149.23	6.309
B	4.735	10	10.64	41.744
C	4.570	10	6.23	149.294
D	4.515	12	5.60	147.977

Table 5.2: Our table contains the score results for our artificial algorithms A, B, C, and D. We colored each cell respective to their score. Bright cells indicate an excellent score, while darker cells indicate bad scores

Then, we can compute the $S_{IWRBS}(\Psi, A)$ for A as described in Equation 5.9:

$$\begin{aligned} S_{IWRBS}(\Psi, A) &= \big(\omega_s * w_s^{-max}(\Psi, D)\big) + \big(\omega_t * w_t^{-max}(\Psi, D)\big) + \\ &\quad \big(\omega_c * w_c^{min}(\Psi, D)\big) + \big(\omega_{sim} * w_{sim}^{min}(\Psi, D)\big) + \\ &\quad \big(\omega_c * w_m^{-max}(\Psi, D)\big) + \\ &= \big(1 * w_s^{-max}(\Psi, D)\big) + \big(1 * w_t^{-max}(\Psi, D)\big) + \\ &\quad \big(1 * w_c^{min}(\Psi, D)\big) + \big(1 * w_{sim}^{min}(\Psi, D)\big) + \\ &\quad \big(1 * w_m^{-max}(\Psi, D)\big) + \\ &= \big(1 * 1.917\big) + \big(1 * 1\big) + \big(1 * 1.053\big) + \big(1 * 1.339\big) + \big(1 * 1\big) \\ &= \underline{\underline{6.309}} \end{aligned}$$

The calculations for B, C, and D are analogous and produce the following IWRBSs (B=41.744, C=149.294, D=147.977).

In Table 5.2, we show the results of NBS, SRBS, WRBS, and IWRBS for all algorithms. We colored each cell according to the best-policy of their approach. The bright color indicates that the score is better than the darker scores of their approach. Based on our example, we would recommend different algorithms with our approaches. Both NBS and SRBS would favor A and B. However, A shows an extremely long runtime, and thus, we can see the weakness of both approaches not weighting outliers correctly. We can also see that WRBS and IWRBS do not recommend the same algorithm as we described before. In summary, we have shown the calculation of all score computations using an example. Since each approach has shown advantages and disadvantages, we cannot estimate which of the approaches gives the best results. The empirical evaluation will show how precise the individual approaches are.

5.3 Summary

To summarize, we introduced our sampling platform that aims to transform the current shared task of product sampling. We started by showing a researcher's use case introducing a new algorithm to the shared task and the problems that arose, which our platform aims to solve. For this purpose, we introduced a general concept and explained all components while adhering to a set of requirements. These

https://doi.org/10.24355/dbbs.084-202009211318-0

requirements ensure the integrity and confidentiality of the platform. Afterward, we separately refined our concept with TIRA to support participant-in-charge software submissions with our sampling framework. The sampling framework controls the execution of each algorithm and ensures a fair measurement of each criterion. Then, we explained the integration of our sampling framework into the architecture of TIRA.

We then introduced the requirements evaluator that computes a set of recommendations for appropriate sampling algorithms based on a given prioritization. We motivated our evaluator by showing a use case of an industrial user searching for an appropriate sampling algorithm and the resulting significant effort. We formally defined four approaches to calculate a score for each algorithm that determines their performance in the sampling process. We developed four score computations: 1) *Nominal-Based Score (NBS)*, 2) *Simple Rank-Based Score (SRBS)*, 3) *Weighted Rank-Based Score (WRBS)*, and 4) *Inverted Weighted Rank-Based Score (IWRBS)*. The first approach NBS aggregates nominal values to compute a score. Our three rank-based score computations rank all algorithms for each criterion individually and aggregates all sub rankings into a score. We finish the section with a detailed example calculation of all approaches.

https://doi.org/10.24355/dbbs.084-202009211318-0

6. Tool Support

In this chapter, we present the tool support for our concepts. We aim is to provide reasonable insight into the implementation and usage of our tool. In Section 6.1, we introduce TIRA and FeatureIDE that support our implementation. Afterward, in Section 6.2 and Section 6.3, we explain the implementation for the concept of our sampling framework (cf. Section 5.1) and requirement evaluator (cf. Section 5.2), respectively. In Section 6.4, we summarize our tool support.

6.1 Building on Existing Tools

In this section, we present the existing tools that we use in our implementation. We aim to explain our intention and motivation for using them.

FeatureIDE

FeatureIDE is an integrated development environment based on Eclipse that assists the user in developing software product lines. It is written in Java, is open-source[1], and is available as prepackaged version[2]. FeatureIDE consists of multiple plug-ins that provide a multitude of visual editors, analyzers, and other tools to ease the development of software product lines for many composers [KPK+17]. Since version *v3.2.0* FeatureIDE releases a library (API) that provides all the functionality we need for our implementation [Fea20]. The API encompasses functionalities such as handling files for feature models in many formats and performing product sampling on feature models. FeatureIDE also provides multiple sampling techniques such as Chvatal, ICPL, IncLing, Random, and YASA. We decided to use the FeatureIDE Library, and thus, to write our sampling framework in Java.

TIRA

TIRA [PGWS19] is a cloud-based web platform that provides evaluation as a service. It is used by several computational science tasks and gives organizers an

[1] https://github.com/FeatureIDE/FeatureIDE
[2] https://featureide.github.io/#download

https://doi.org/10.24355/dbbs.084-202009211318-0

easy and reliable solution for performing participant-in-charge software submissions [TIR20]. Users can access TIRA via its Website[3] and browse available tasks in the task market. The process of hosting a task on TIRA is not automatic and requires an application to the TIRA team. When accepted, the organizer gains access to a set of VMs which he can distribute to various participants. Each participant can manage their VM with full privileges. Only when executing the evaluation with a VM, it gains isolated access to the test data. Isolation means that the VM is not in control of the participant, but instead, it is controlled by TIRA itself. Hence, the test data is not shared with the participants but can still be used by their submitted software. For us, an easy and reliable solution to host tasks, free environments for submitted software, and secured test data were the key factors for deciding on TIRA for our implementation.

6.2 Sampling Framework

We introduce the concept for our sampling framework in Section 5.1 as a solution to the uncoordinated shared task (cf. Section 3.1) of product sampling. In Section 6.2.1, we shortly provide an overview of the important factors that we need to consider for the implementation. Then, in Section 6.2.2, we explain the architecture of our sampling framework in detail. Afterward, in Section 6.2.3, we explain how to use our sampling framework by showing use-cases such as adding a new solver, starting an evaluation, and more. In Section 6.2.4, we explain how to integrate our framework into TIRA to evaluate sampling algorithms automatically.

6.2.1 Overview

Our sampling framework aims to automate the process of comparing t-wise sampling algorithms. The general idea is that users only need to execute the framework on their sampling algorithm. The framework computes and represents the results in such a way that we can directly compare them to the results of other algorithms. We choose the name *sampling framework* as our software handles the registered sampling algorithms of a user with the *Hollywood* principle, i.e., the framework calls the sampling algorithms not the other way around.

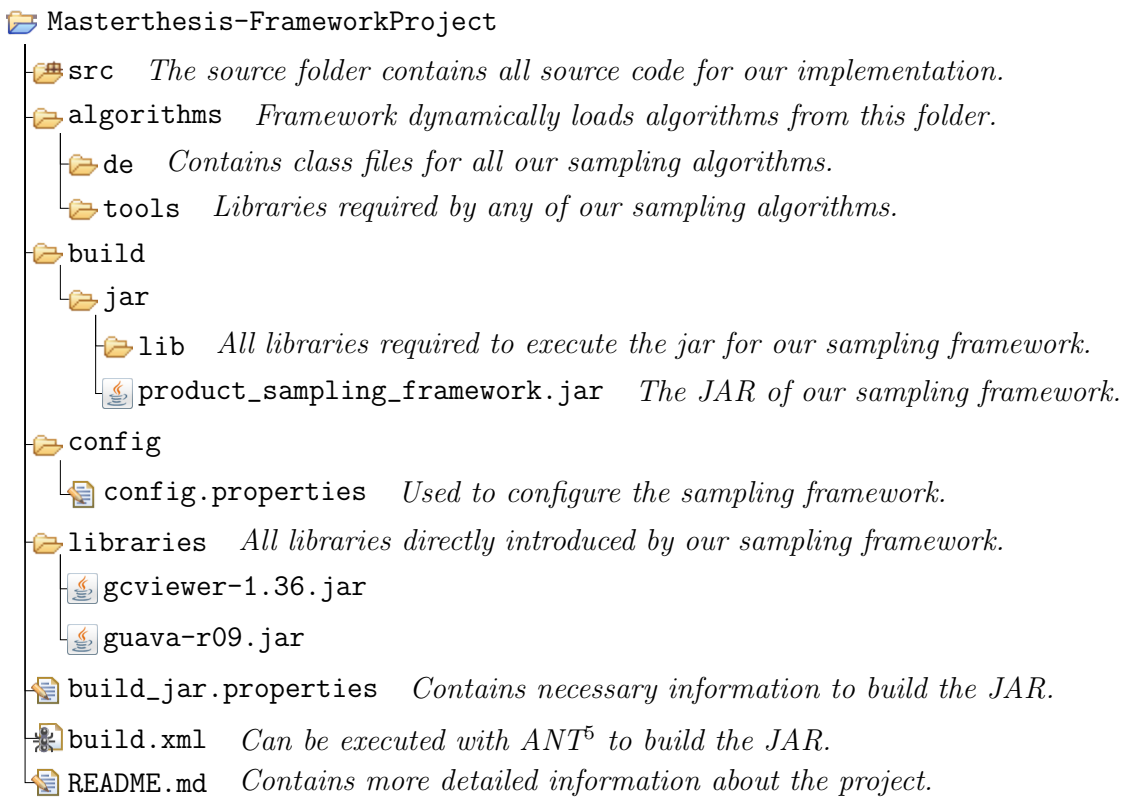
Project Setup

Introducing the project and file structure for our implementation serves two purposes. First, it gives a great overview of the different files of the project and explains their existence. Second, it provides fundamental insight on how to continue the development of the sampling framework. Our implementation is available on a GitHub[4]. In the following, we explain the structure of our project:

[3] https://www.tira.io

[4] https://github.com/Subaro/Masterthesis-FrameworkProject

https://doi.org/10.24355/dbbs.084-202009211318-0



The first step is to download the sampling framework from our repository to start developing. Afterward, import it in any IDE that supports Java development, e.g., Eclipse. All the required dependencies are available in the project. Adding all dependencies from the *build/jar/lib* folder finishes the setup. Bear in mind, that the *build.xml* requires FeatureIDE to be downloaded and configured in the *build_jar.properties* as it automatically compiles the FeatureIDE library followed by the JAR for the sampling framework.

General Process

We show an activity diagram for the general process of the sampling framework in Figure 6.1. The first important step is the right configuration of the sampling framework. For this purpose, we provide a configuration file next to the framework, which offers options for the configuration of the evaluated algorithms, their aimed t-wise coverage, the timeout, and more. In general, the file facilitates the setup of the framework. Next, we load all models from the input path, which can be freely configured, and the algorithms specified in the configuration file from the algorithms folder (cf. Project Setup). Now, we start evaluating all algorithms for each model. For that, we select the first pair of model and algorithm and prepare them for their execution. We start the sampling algorithm as a separate Java process that computes the sample for the model.

Meanwhile, we monitor the runtime for the sampling process. Not every algorithm computes the samples for the same format, so users need to parse their sample for our framework. Additionally, users measure memory data such as memory consumption and parse them as well. We compute statistics such as sample coverage and validity based on the generated sample, measured runtime, and measured memory data.

https://doi.org/10.24355/dbbs.084-202009211318-0

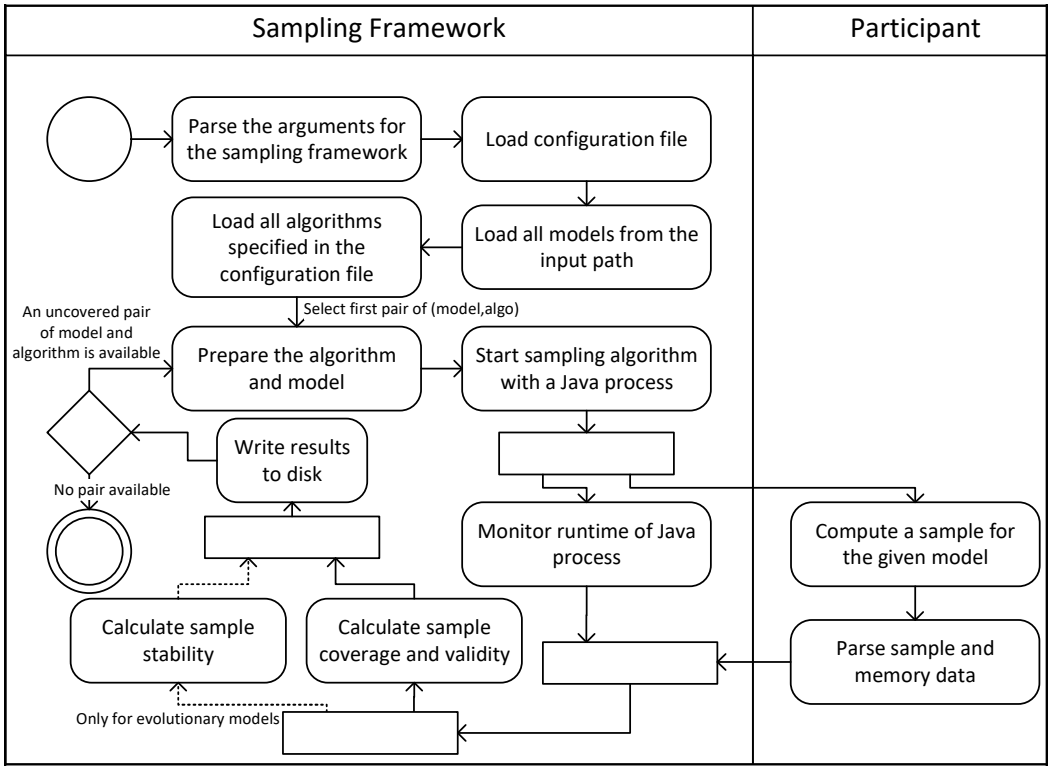


Figure 6.1: The activity diagram describes the general process of the sampling framework. We divided the process into activities performed by the framework and the implementations of a participant

When the input models represent the changes of one product line over time, the framework automatically computes the sample stability. At last, we write the results into a file. The framework repeats the same process until it processed all algorithms for each model. Then the framework terminates.

6.2.2 Architecture

The FeatureIDE benchmark library[6] is a general framework for benchmarking various algorithms and serves as the template for our sampling framework. It generally controls the execution process for algorithms, provides more comfortable FeatureIDE API usage, redirects the software output to the framework, handles the writing of output data, and supports the reading and writing of configuration files. We forked the benchmark library and extended it to work solely for the automated comparison of sampling algorithms. In Figure 6.2, we show the architecture of the sampling framework as a simplified class diagram. For brevity, we omitted most of the attributes, methods, or associations and focused on the most important ones. Now, we explain all classes package-by-package in detail. For brevity, we reference packages such as `de.ovgu.featureide.sampling.logger` just as a `logger`.

[6]https://github.com/skrieter/FeatureIDE_Tools/tree/master/de.ovgu.featureide.fm.benchmark

https://doi.org/10.24355/dbbs.084-202009211318-0

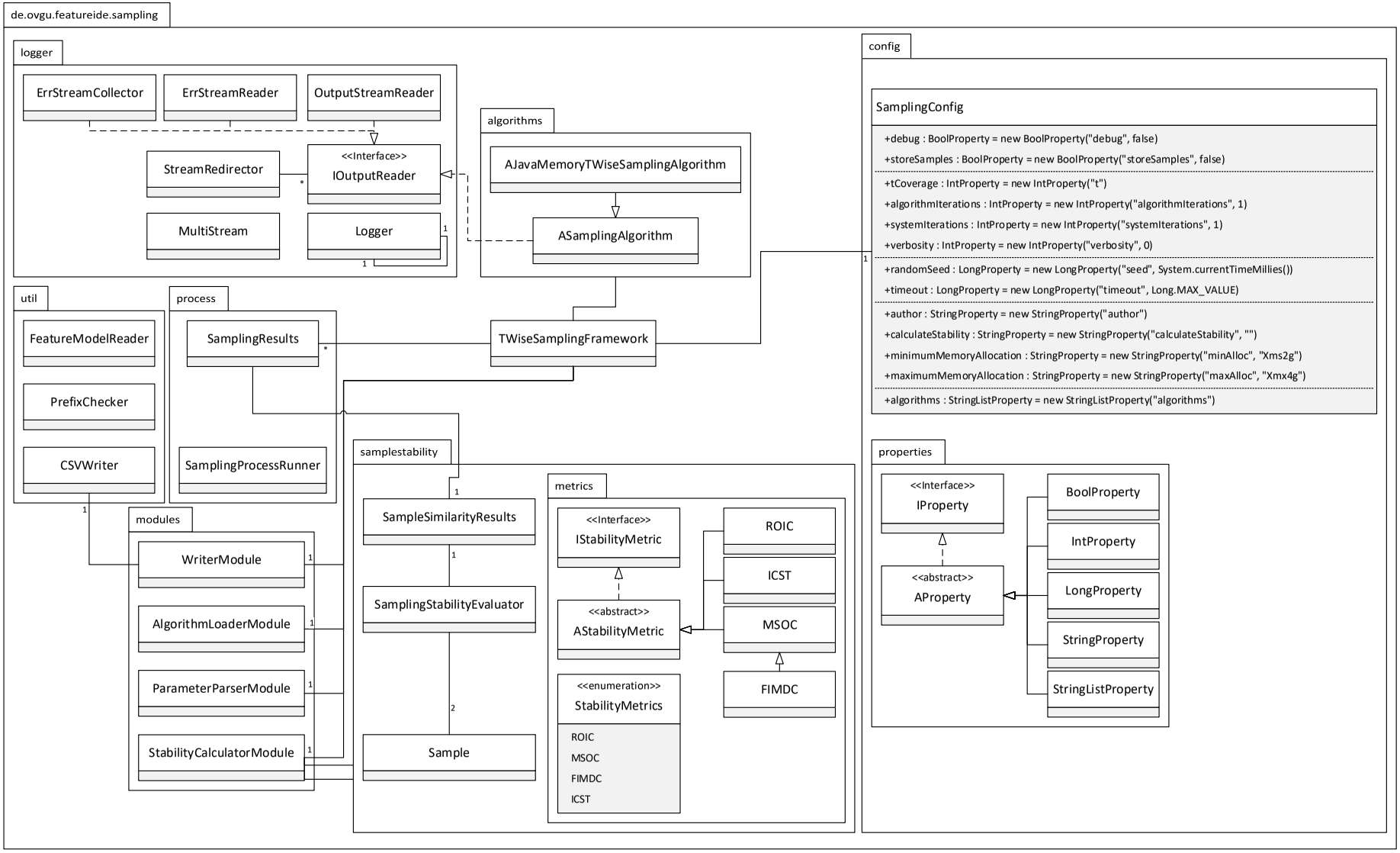


Figure 6.2: Shows the class diagram of the sampling framework architecture. For brevity, we omitted attributes, methods, and other small details.

https://doi.org/10.24355/dbbs.084-202009211318-0

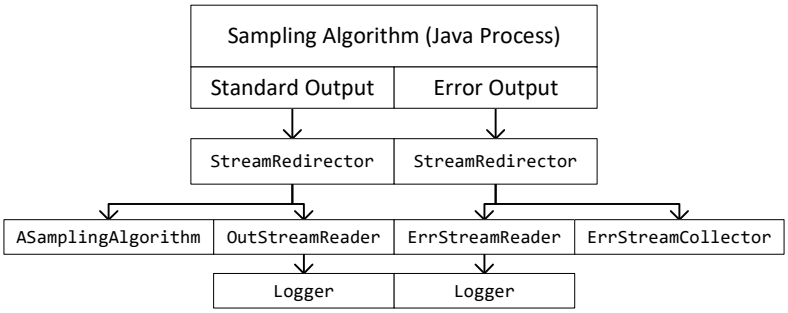


Figure 6.3: Handling of standard and error output of sampling algorithms executed as Java processes

Package: `logger`

The `logger` provides a logger that can be accessed statically to log information or errors to our console and persistently to our log files. We use it to to visualize the framework's progress and to inform the user whenever an error occurs. To do so, we implemented the class `Logger` with the singleton design pattern [GHJV95]. The logger provides convenient methods such as `Logger.getInstance().logInfo("Test")` to log information or errors which it persistently saves into a specified log file. Additionally, it is possible to provide the level of verbosity for each log. The verbosity can be configured at the start of the framework to control the amount and subtlety of the shown logged information.

The framework creates four log files on start, namely `console_log`, `console_log_reduced`, `error_log`, and `error_log_reduced`. The files `console_log` and `console_log_reduced` contain all information that the framework shows to the console. The file `console_log` contains all logged information with a verbosity equal to or less the value set in the framework configuration, while `console_log_reduced` only shows logs with a verbosity of zero (i.e., the most basic/important information). The same behavior applies to `error_log`, and `error_log_reduced` but instead of showing logged information, they only contain logged errors. The MultiStream assists the `Logger` by streaming each log call to each respective file and the standard output of the framework.

One of the most important points is that the framework also logs errors or information from the participant's sampling algorithms. However, as these algorithms are independent of our framework, they can also not use our logger. For that, we constructed streams, as shown in Figure 6.3. When starting the Java process, we register one `StreamRedirector` on the standard output and one on the process's error output. A `StreamRedirector` redirects an input stream to one or more output streams. In our case, one directs the standard output to the algorithms interface `ASamplingAlgorithm` so that a user can react if wanted and also to the OutStreamReader, which invokes the `Logger`. The second `StreamRedirector` redirects the error output to the `ErrStreamReader` that invokes the `Logger`. It also redirects errors to the `ErrStreamCollector`, which collects all errors, and thus, helps to check if any error occurred after the sampling algorithm finishes.

https://doi.org/10.24355/dbbs.084-202009211318-0

Package: `utils`

This package contains multiple classes that provide useful functionalities while not fitting in any specific package.`FeatureModelReader` provides a simple way of loading feature models from a given file, folder, or even ZIP file. We implemented the `PrefixChecker` to compute the longest common prefix of a given set of input strings. Last but not least, the `CSVWriter` that makes it easier to write data into CSV files. The class descriptions provide further information about the usage of the class and how to use them properly.

Package: `config` & `config.properties`

Having a configuration for the framework makes it easier to change its behavior according to different situations. Use-cases such as switching from pairwise-coverage to 3-wise coverage, changing the algorithm that we want to evaluate, or calculating the stability between samples are possible scenarios we need to consider. For that, we implemented `IProperty` as an interface for all possible types of configuration options. For our framework, we need properties of type `boolean`(`BoolProperty`), `int`(`IntegerProperty`), `long`(`LongProperty`), `String`(`StringProperty`), and a list of `String`(`StringListProperty`). Each property consists of a key and optional default value. The `SamplingConfig` represents the configuration for our framework in the memory. Further, it provides methods to read configuration from files and save the current configuration into a file. For that, it needs to know each property that automatically registers themselves when initiated. In the following table, we shortly explain each configuration option:

Option	Default	Description
`debug`	`false`	When `true` removes all temporary files after the framework finishes.
`storeSamples`	`false`	When `true` persistently saves all computed samples.
`tCoverage`	`2`	The degree of t-wise coverage to compute.
`algorithmIteration`	`1`	The number how often the computation with an algorithm should be repeated.
`systemIteration`	`1`	The number how often the computation of a system should be repeated.
`verbosity`	`0`	Determines the verbosity level for logged entries.
`randomSeed`	-	The seed used for randomizing the models before the evaluation.
`timeout`	`MAX`	The timeout for each evaluation.
`author`	-	Identifier for the creator of the evaluated sampling algorithm.
`calculateStability`	`false`	When `true` calculates the sampling stability.
`minimumMemoryAllocation`	`Xms2g`	The minimum amount of memory allocated for the sampling process.
`maximumMemoryAllocation`	`Xmx4g`	The maximum amount of memory allocated for the sampling process.
`algorithms`	-	List of all algorithms evaluated in this framework run.

Package: `samplestability` & `samplestability.metrics`

This package provides the functionality to compute the stability of samples. For that, we use the concept and implementation of Pett [Pet18a]. The `metrics` package contains all metrics for sample stability introduced by Pett [Pet18a]. The `SamplingStabilityEvaluator` computes all metrics for two given samples. For that, it uses the `Sample` class, a data structure for samples required for calculating the stability metrics, that provides some handy functionalities, e.g., omitting deselected features. The framework caches the results for each metric in `SampleSimilarityResults`.

https://doi.org/10.24355/dbbs.084-202009211318-0

Package: `algorithms`

The abstract class `ASamplingAlgorithm` provides a general interface for participants to make their sampling algorithm executable by our framework. This interface is generally applicable to all kinds of software that one can execute via a bash command. The participants need to provide a bash command to start their sampling algorithm. At the same time, information such as the path to the current model, the path to the output file, and the degree of t-wise coverage are available in the interface. It is also the participant's responsibility to implement the parsing of their sample into the data structure required by the framework. As computing the memory statistic for all kinds of possible programs is not easy, we provide a possibility to compute them themselves and parse them into a data structure accepted for our framework.

Computing memory statistics seems impossible for all kinds of programs. However, all of our sampling algorithms are written in JAVA, and thus, we implemented `AJavaMemoryTWiseSamplingAlgorithm`, a more restricted interface that automatically computes memory statistics based on garbage collection logs. For that, we extend the command for each algorithm such that garbage collection logs are created. We use GCVIEWER[7], an open-source tool that visualizes verbose GC output generated by Sun / Oracle, IBM, HP and, BEA Java Virtual Machines. We use it as a library to analyze garbage collection logs to compute memory statistics. The most important statistic that we extract is the total number of created bytes.

Package: `process`

The `SamplingProcessRunner` prepares and controls the execution of the JAVA processes for each evaluated sampling algorithm. Before starting the process, it is vital to set up the output stream, as described in the `logger` package. Also, we need to request the command from the `ASamplingAlgorithm` interface to start the process accordingly. After the process terminates, the `SamplingProcessRunner` collects all results such as runtime, sample, memory statistics, validity, and if no error occurred during the sampling process. The framework saves all information in the data structure `SamplingResults` and forwarded to the `TWiseSamplingFramework`.

`TWiseSamplingFramework` **and the** `modules` **Package**

The class `TWiseSamplingFramework` is the entry point for our framework and also controls its complete execution. We created four modules that each has its responsibilities. The first module is the `ParameterParserModule` that, as the name implies, parses the arguments given to the sampling framework. They are used to quickly override some of the configuration options without having to edit the configuration file. Further, they are used to specify the input and output path for the framework. In the following, we shortly introduce all supported arguments:

[7] https://github.com/chewiebug/GCViewer

https://doi.org/10.24355/dbbs.084-202009211318-0

Parameter	Description
-in *path*	Sets the input path where all models should be loaded from.
-out *path*	Sets the output path where all results should be written to.
-alg *algorithm*	Overrides the value of the `algorithms` property to *algorithm*.
-t *integer*	Overrides the value of the `tCoverage` property to *integer*.
-store *boolean*	Overrides the value of the `storeSamples` property to *boolean*.

The second module, the `AlgorithmLoaderModule`, loads all algorithms for this run. As said before, the `algorithms`'s property in the configuration file specifies the evaluated algorithms. For example, if we implemented the `AJavaMemoryTWiseSamplingAlgorithm` for the Chvatal algorithm as `de.ovgu.featureide.sampling.algorithms.Chvatal` then, we need to add the full-qualified class name to the `algorithms` property (i.e., `algorithms=de.ovgu.featureide.sampling.algorithms.Chvatal`). Basically, the module creates a `ClassLoader` for the *algorithms* folder (cf. Project Setup) and tries to load all configured classes as `ASamplingAlgorithm`. For that to work, the user needs to compile his interface and drop it into our *algorithms* folder. The module also produces errors if it cannot find any configured classes so that the user can act accordingly.

The `StabilityCalculatorModule` computes the stability between samples. We introduced this as an optional feature as it makes only sense for models that represent the same product line over time. The user can generally activate or deactivate the feature by setting the configuration option `calculateStability` to `true` or `false`. However, if the `calculateStability` option is not mentioned in the configuration file, then the default value indicates the framework to decide whenever to compute samples or not automatically. This is decided with the `PrefixChecker` of the `utils` package. Whenever all input models share a prefix with the length of at least five characters, we assume that the stability should be calculated. Computing the stability requires the samples to be cached so that we can compute their similarity. The stability module caches the least possible amount of samples that are necessary for the computation. The stability of each sample is computed after the process runner passes the sampling results to the `TWiseSamplingFramework`.

The framework invokes the fourth module, the `WriterModule`, after each evaluation iteration, which writes all results into a CSV file called *data.csv*. For instance, in Figure 6.4, we show the output path's structure after the sampling framework terminates. When the framework is configured to store all computed samples, the module also generates a folder named *samples* and one subfolder for each evaluated system. Each subfolder contains the samples generated by all evaluated algorithms for this system.

6.2.3 Use-Cases for the Product Sampling Framework

Now, we present an example project to explain how to use our framework. It aims to help everyone who wants to evaluate their own algorithm or just want to execute our framework. We show the structure and content of the example project in Figure 6.5. For brevity, we generally omit trivial or unnecessary information. In the first step,

https://doi.org/10.24355/dbbs.084-202009211318-0

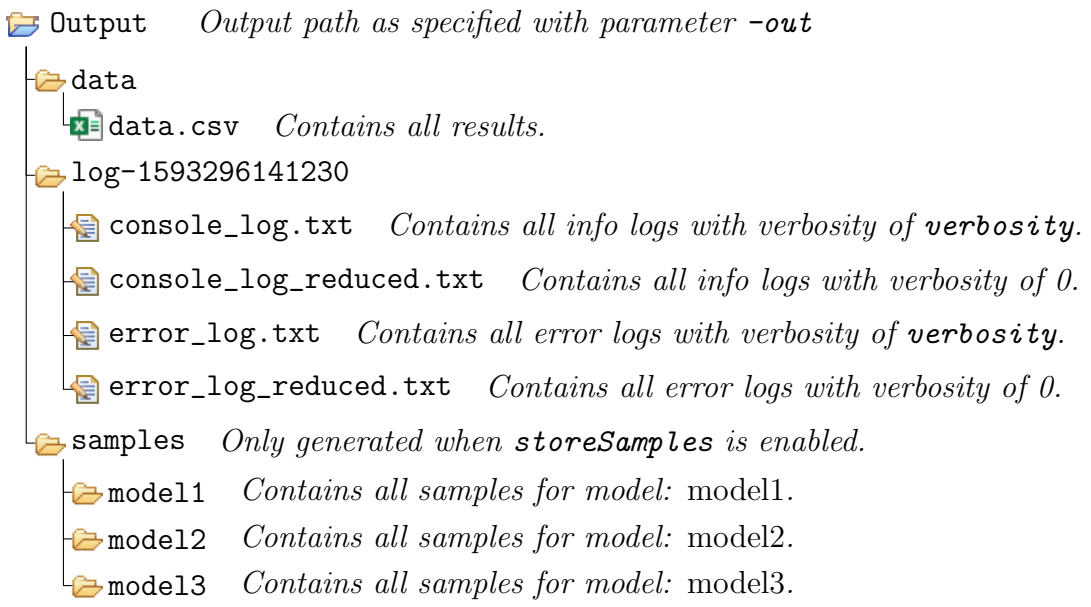


Figure 6.4: The structure and files of the sampling framework output folder

the user downloads the JAR of the sampling framework and all necessary dependencies (*lib* folder). For our example, we consider that the user wants to evaluate a new sampling algorithm on a set of self-provided models. For that, the user prepares the folder `MyInputModel` that contains his models. Also, for the output, the user creates the folder `MyOutput`. Now, the user implements his sampling algorithm and bundles the implementation of his sampling algorithm as `MySampler.jar` that contains two classes. The class diagram in Figure 6.5 shows the `MySampler` class, responsible for computing the sample of a given DIMACS model. The second class `MySamplerUtils` contains the utility class to transform the computed sample into the data structure required by us. Now, the user stores his sampler in the *algorithms/tools* folder where it is available for the algorithms interface. His implementation of the `ASamplingAlgorithm`, called `MySamplingAlgorithm`, overrides the method `addCommandElements()` to tell the framework how to invoke his sampling algorithm and `parseResults()` to parse the output of his algorithm into the format required by us. For the next step, the user creates a configuration file and configures the framework to his liking. It is important to provide the full-qualified class name of his interface. For our example, the user-created the `MySamplingAlgorithm` inside the JAVA default package, and thus, only provides the class name of his interface. In the last step, the user creates the *SamplingFramework.sh* launch script and contains: 1) a name of a valid configuration file, 2) an absolute path to the input folder, and 3) an absolute path to the output folder.

6.2.4 Integration with TIRA

In this subsection, we present how to combine our sampling framework with TIRA. For that, we connect to the example before. Instead of running the sampling framework via *SamplingFramework.sh* directly, the user wants to run the framework on TIRA. The first thing he needs is a valid login for an assigned virtual machine. With that login, he can access the virtual machine via SSH and store the project on

https://doi.org/10.24355/dbbs.084-202009211318-0

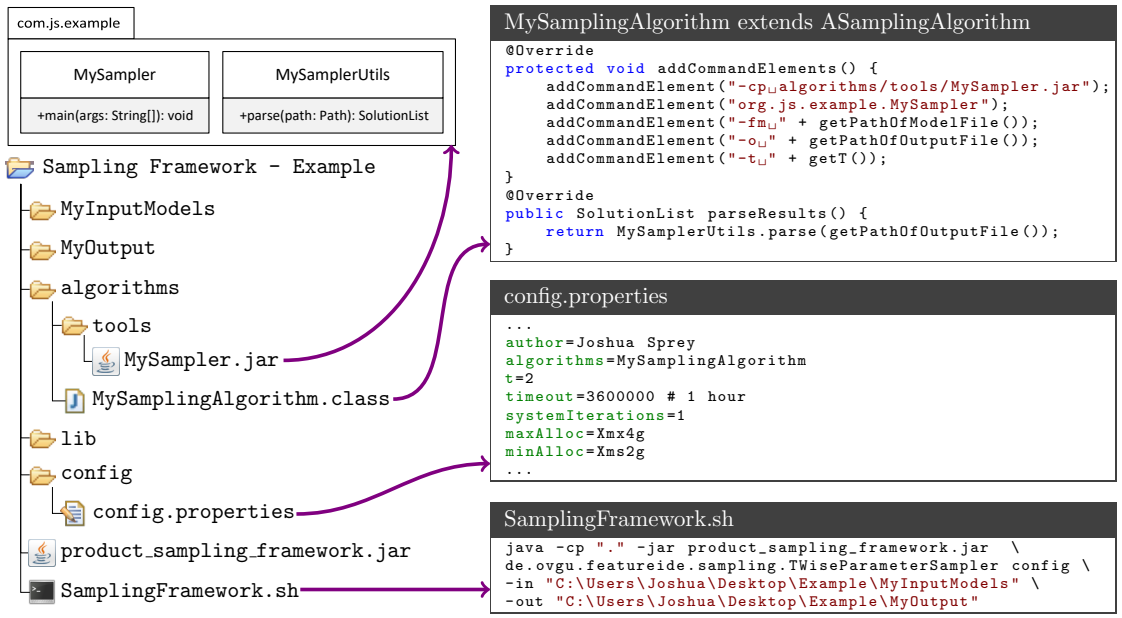


Figure 6.5: The structure and content of a sampling framework example

the device. Then, in the user interface of TIRA, he can add one or more software for the evaluation. The configuration of the software, as shown in Figure 6.6, consist of the following entries:

- Command: The command TIRA uses to execute the software. In our example, the user edits the starting script *SamplingFramework.sh* in such a way that it requires a parameter for: 1) the output directory, 2) the input directory, and 3) the degree of t-wise coverage. TIRA provides a set of variables that are automatically resolved when staring the software such as the output directory (`$outputDir`) or the path to the input dataset (`$inputDataSet`).
- Input dataset: The data package that should be evaluated for this run.
- Input run: The results of a previously executed software. Can be used for iterative techniques for example.
- Working directory: The directory where TIRA executes the command to start the software.

By clicking *Run*, it is possible to execute the software, and TIRA starts to sandbox the virtual machine. It is not possible to configure or start new software while TIRA sandboxes a virtual machine. Instead of the regular UI to configure a software, it shows the console output of the executed software. This behavior helps identify problems in the execution without having to wait for the whole process to finish. When the software finishes it's execution, then TIRA provides a list of all available results computed by our virtual machine. In Figure 6.6, at the bottom, we show a list of calculated results that can be: accessed for detailed information (ℹ), downloaded (⬇), or deleted (⊗). These results are currently not published. For that, we need to evaluate the results with a so-called *Evaluator* to publish them. We aim to use our requirements evaluator that we introduce in Section 5.2 as TIRA 's *Evaluator* to

https://doi.org/10.24355/dbbs.084-202009211318-0

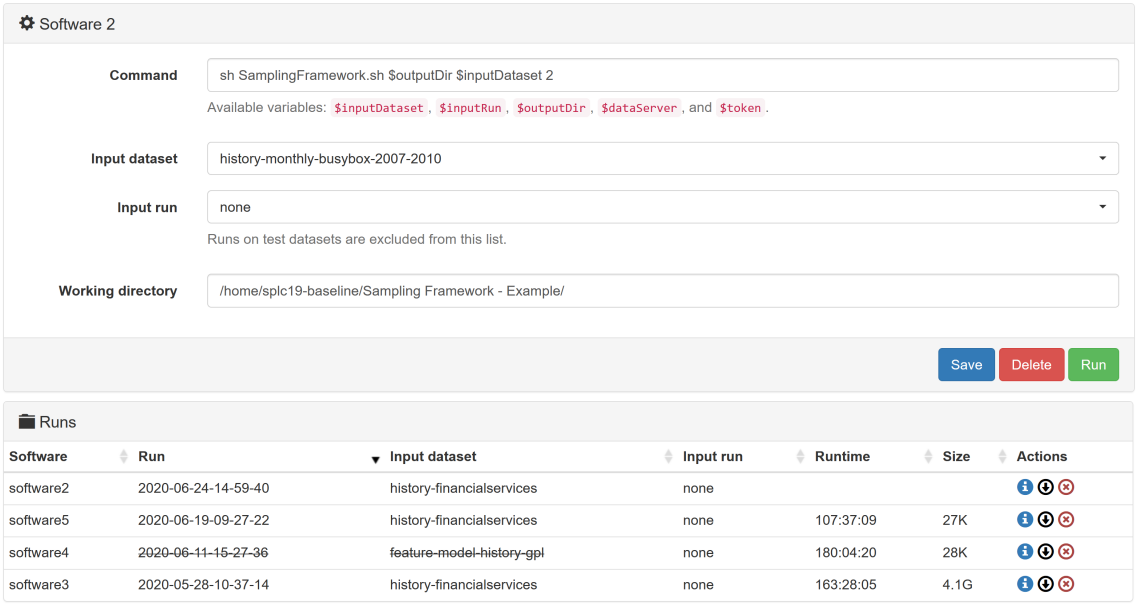


Figure 6.6: The configuration of a software and the results of previous executions in the TIRA UI.

publish our results. The evaluator is responsible to compute our score computations and to recommend sampling algorithms to the user.

6.3 Requirements Evaluator

In this section, we present the implementation of the requirements evaluator that we introduce in Section 5.2. As said before, we aim to implement the requirements evaluator to be used as an *Evaluator* in TIRA. Following best practices for TIRA, we decided to use the script language Python for the implementation.

6.3.1 Score Calculation with Python

The general process for the evaluator starts by loading the results produced by the framework. We use the *pandas*[8] framework to load, manipulate, and reshape our data in the whole process. The main reason for using *pandas* is that it is built for easy, powerful, and flexible data analysis. The functionalities provided by *pandas* seem a bit over the board when only considering to compute the scores that we introduce in Section 5.2. Especially when comparing the performance of *pandas* against native array and list handling. However, the performance for the score computation is linear to the number of participating algorithms, and thus, not a deciding factor for our implementation. The further point is that we need to visualize our results for the empirical evaluation of our thesis. Hence, we decided to construct a simple graphical interface to visualize the data for the computed scores and even to compare the results of the sampling framework. Functionalities such as easy sharing of data, creating views on data, and filtering with masks are helpful. All these reasons contribute to the decision to select *pandas* for our data handling.

[8] https://pandas.pydata.org

https://doi.org/10.24355/dbbs.084-202009211318-0

The evaluator computes all scores for each participating algorithm. The calculation for the score proceeds in the following steps:

1. Load all data with *pandas* from the input path (each data is a `data.csv` generated by the sampling framework).
2. Compute for each evaluated iteration the nominal values required by the NBS (cf. Equation 5.1) score.
3. Compute averages for all nominal values and all evaluation criteria (i.e., sample size, sample runtime, ...).
4. Compute the NBS (cf. Equation 5.1), SRBS (cf. Equation 5.5), WRBS (cf. Equation 5.8), and IWRBS (cf. Equation 5.9) for all algorithms.
5. Determine the score ranking for all algorithm depending on the score, i.e., the NBS Rank shows the ranking based on the NBS score, while the SRBS Rank shows the ranking based on the SRBS score.

We can present the the resulting scores, ranks, and averages in different ways. For that, we implemented three modes: console mode, TIRA mode, and the GUI mode. For that, we need parameters to specify the necessary information. The evaluator accepts the following parameters:

Parameter	Default	Description
`--in PATH`	-	Path to the input folder that contains all `data.csv` files. It is not required for each to be named `data.csv`. They just need to be CSV files.
`--out PATH`	-	Path to the output folder [TIRA and GUI mode only].
`--size NUM`	1	Changes the sample size prioritization to NUM.
`--runtime NUM`	1	Changes the sample time prioritization to NUM.
`--coverage NUM`	1	Changes the sample coverage prioritization to NUM.
`--similarity NUM`	0	Changes the sample similarity prioritization to NUM.
`--memory NUM`	0	Changes the sample memory prioritization to NUM.
`--gui`	-	When given, starts the evaluator in GUI mode.
`--tiraInput PATH`	-	When given, starts the evaluator in TIRA mode. The path aims to the folder containing the `data.csv` that should be evaluated.

Evaluator: Console Mode

The console mode is based on the idea to generate the scores with a given prioritization and to show the results directly in the console. Only the `--in` parameter is required to specify the input folder.

Evaluator: TIRA Mode

The idea of this mode is to use a TIRA *Evaluator* to compute all scores for an evaluated run (i.e., only one algorithm at a time). To do so, we can select any valid software run in TIRA and evaluate them with our requirements evaluator. For instance, all the evaluated runs shown in Figure 6.6 at the bottom can be given as input to an *Evaluator* in TIRA. Figure 6.7 shows the UI to start an *Evaluator* based on a selected input run. The *Evaluator* could generally be any program that accepts

https://doi.org/10.24355/dbbs.084-202009211318-0

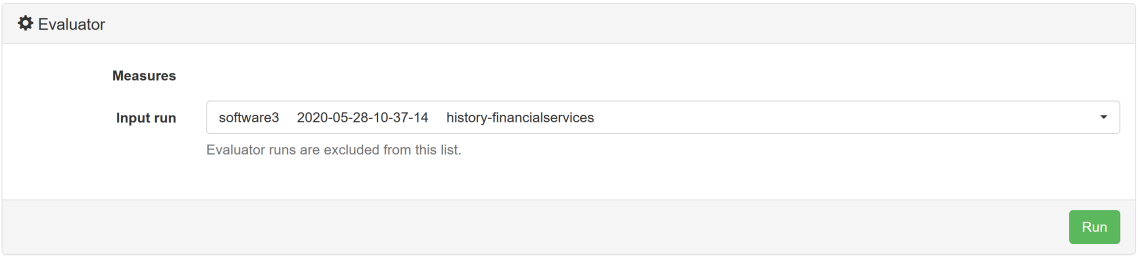


Figure 6.7: The UI to start an Evaluator in TIRA with a selected run as input

the following parameters: 1) folder that contains the ground truth, 2) folder that contains the currently selected input run, and 3) the output folder. The main task for the *Evaluator* is to generate a file called `evaluation.prototext` in the output folder. This file should contain all metrics that the *Evaluator* should calculate. In our case, we use our requirements evaluator Python script as an *Evaluator* in TIRA. However, we faced some technical restrictions as TIRA is still in development. First, TIRA does currently not support to configure the evaluator as freely as a software. In the end, our requirements evaluator could only be installed by the TIRA team themselves. Second, starting an *Evaluator* with more than one input run is also not supported. However, we require all input runs to calculate our ranking-based approaches. Together in cooperation with the TIRA team, we decided to place the results for all input runs beside the ground truth so that the evaluator could access them. The parameter `--in` is set to the folder that contains the ground truth and all results. As we evaluate only one run at a time, we need a second input parameter named `--tiraInput`. This parameter contains the outputs of a sampling framework run (cf. Figure 6.4). Now, the evaluator searches the `data.csv` recursively and uses the content to determine the currently evaluated algorithm. Finally, we compute all scores usually, but only write the results for the algorithm of interest in the `evaluation.prototext`. TIRA uses the file to generate user-friendly results lists. With this implementation, it is possible for us to evaluate automatically and to compare sampling algorithms. However, the technical restriction still requires manual effort that needs to be performed by the TIRA team.

Evaluator: GUI Mode

Using *tkinter*[9], the de-facto standard GUI package for Python, we developed a simple but efficient GUI to visualize the sampling framework's results and their scores. With the GUI, we can compare all algorithms based on their sampling size, time, coverage, similarity, and memory. Further, we can interactively compute scores for a single prioritization, a list of prioritization, or a range of prioritization. We support an automatic export of all plots, which is helpful and saves effort in writing plotting scripts all over again. In Figure 6.8, we show the GUI mode of our requirements evaluator. In the following, we explain the different groups of control.

Prioritizations enable the user to create plots for three kinds of use cases to visualize the score computation. The first use case computes only the results for one prioritization. The resulting scores for each approach are visualized as bar plots where

[9]https://wiki.python.org/moin/TkInter

https://doi.org/10.24355/dbbs.084-202009211318-0

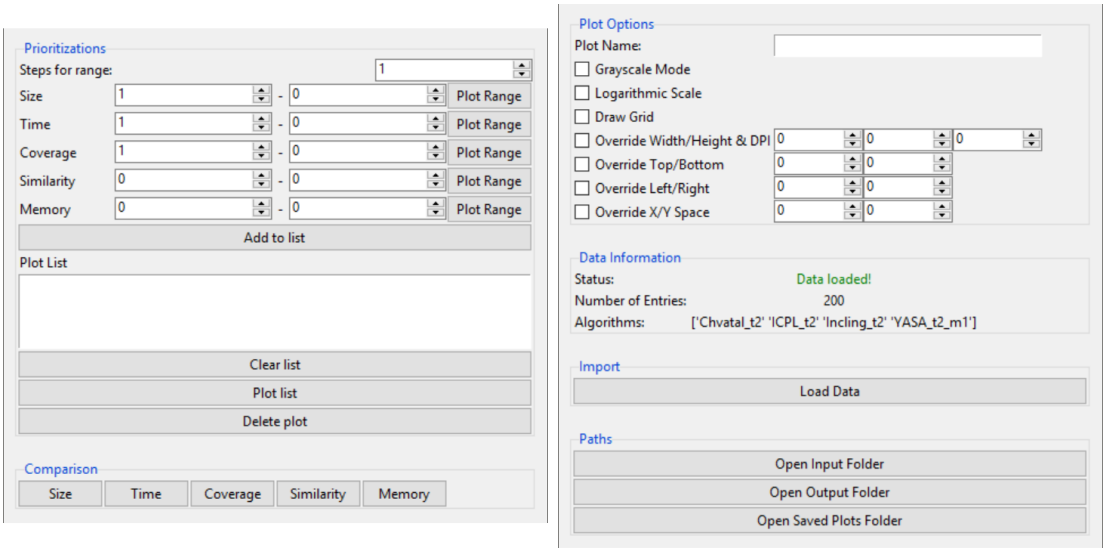


Figure 6.8: The graphical mode for the evaluator allows a multitude of functionality to visualize the results for the automated evaluation of sampling algorithms

each bar represents the score for one algorithm. The second use case is computing a list of specific prioritizations. The user can add the current prioritization as defined on the top left to the plot list. Clicking *Plot list* creates four scatter plots, one for each score approach, having the different prioritizations on the x-axis and the values for each algorithm on the y-axis. The third use case computes the scores for a static prioritization where only the priority for one criterion is defined as a range. For example, the user is interested in how the score for the evaluated algorithm behaves when the sampling size priority increases. The user can define the range and increasing steps. The four resulting scatter plots, one for each score approach, contain the variable criteria priority on the x-axis and each algorithm's score on the y-axis. The program automatically exports the plot when the user provides a name in the plot options.

Comparison enables the user to create scatter plots that compare the measured sampling framework data of each algorithm. The resulting scatter plots show the number of features on the x-axis and the criteria's values on the y-axis. The program automatically exports the plot when the user provides a name in the plot options.

Plot Options provide a multitude to change the visual output of plots such as the resolution of the plots if the resulting plots should contain a grid or show the results on the y-axis in the logarithmic scale. Another important field is the Plot Name that sets the name for all exported plots. The application tags each exported plot by timestamp to prevent name conflicts.

Data Information and Import are responsible for loading from the input path specified with the `--in` parameter. Further, it shows the number of evaluated runs and the participating algorithms.

Paths provide buttons that open the *input*, *output*, and *export* path in the editor.

https://doi.org/10.24355/dbbs.084-202009211318-0

6.4 Summay

To summarize, we started by explaining the implementation of our sampling framework. At first, we introduced TIRA and FeatureIDE that we use for our implementation. We, then provided an overview of our project structure, the general process, and the architecture of our sampling framework. To help potential users, we introduced and explained by an example of how to add a new sampling algorithm to the framework and the required steps to execute the framework. We also covered the integration of our sampling framework into TIRA to automatically compute sampling algorithms. Afterward, we introduced the implementation of the requirements evaluator. It is responsible for calculating all of our score approaches based on the data generated by the framework. We covered the evaluator's general process, how to use it, and its integration into TIRA. Further, we introduced the graphical mode of the evaluator to visualize the data produced by the framework and the score computation.

https://doi.org/10.24355/dbbs.084-202009211318-0

7. Evaluation

Configurable systems can offer many benefits if the variability is handled properly [DMTR08]. Testing such a system is complex and costly and is still considered a challenge in the quality assurance process [McG01, MKR+16]. Product sampling is a modern technique to reduce the number of products to be tested. More than 50 techniques (cf. Section 4.2) for sample calculation were presented. We have developed a sampling framework (cf. Section 5.1) to compare these techniques automatically. Furthermore, we developed a requirements evaluator (cf. Section 5.2), which compares the algorithms and makes recommendations for specified requirements.

In this chapter, we perform an empirical evaluation on more than 160 real-world feature models of varying sizes to find out if our concepts produce meaningful results and if we can answer some of our questions about sampling algorithms. We start in Section 7.1 with the introduction of our research questions. In Section 7.2 and Section 7.2, we introduce the setup for our evaluation and describe the individual experiments we perform. Subsequently, we present the results for the comparison of sampling algorithms in Section 7.4 and discuss them in detail. Then, in Section 7.5, we present and discuss the results for the ranking of sampling algorithms based on the score calculation. Afterward, in Section 7.6, we explain all threads of validity and summarize the evaluation in Section 7.7.

7.1 Research Questions

We decided to split the research questions into two areas. The first area consists of research questions about comparing sampling algorithms concerning their evaluation criteria such as sample size or sampling runtime. Evaluation criteria are often subjects of evaluations for sampling algorithms as the results can be compared to the results of other sampling algorithms. The most common criteria used for evaluations are sample size (cf. column sampling efficiency in Table 4.1), sample time (cf. column testing efficiency in Table 4.1), or coverage (cf. column effectiveness in Table 4.1). At the same time, criteria such as memory consumption and sample

https://doi.org/10.24355/dbbs.084-202009211318-0

similarity are rarely used for evaluations. We aim to compare all algorithms with regard to each evaluation criteria by the following research questions:

RQ$_1$ Which t-wise sampling algorithm calculates the sample fastest?

RQ$_2$ Which t-wise sampling algorithm computes the smallest samples?

RQ$_3$ Which t-wise sampling algorithm achieves the highest t-wise coverage?

RQ$_4$ Which t-wise sampling algorithm consumes the least memory in the process?

RQ$_5$ Which t-wise sampling algorithm calculates the most similar samples?

The second area of research questions focuses on the score computation for sampling algorithms. We introduced four approaches to calculate the score for a sampling algorithm. We aim to determine whether each approach calculates more precise recommendations. For that, we defined the following research questions:

RQ$_6$ Which score computation computes the correct algorithm for only one requirement?

RQ$_7$ Which score computation computes the most precise recommendation for a given prioritization?

7.2 Setup

Understanding the setup and experiments is essential to understand the results of our evaluation. Further, it supports other users when reproducing our work. We aim to introduce our machines, used software, evaluated systems, and our experiments.

Machine & Software Specification

We performed all our experiments on a machine with the following specification:

- **OS** CentOS Linux 7, 64-bit
- **CPU** Intel Core Broadwell Processor with 16 sockets which each has a core at 2.4 GHz clock rate.
- **Memory** 64 GByte

In the following, we list all used software and their specific version:

- **Java** OpenJDK version: `1.8.0_232`
- **FeatureIDE-Library** Compiled from branch `config_generation`[1]
- **TIRA** No specific version information available. Website[2] accessed from February to August 2020.
- **Chvatal & ICPL** Implementation from SPL Covering Array Tool (SPLCAT) Manifest Version: `1.0`
- **IncLing & YASA** Implementation from FeatureIDE-Library
- **Python** Version: `3.7.1`, Modules: `pandas==1.0.4`, `matplotlib==3.1.1`, `tkinter==8.6`

[1] https://github.com/FeatureIDE/FeatureIDE/tree/config_generation
[2] https://www.tira.io

https://doi.org/10.24355/dbbs.084-202009211318-0

Feature Model Name	#Features	#Constraints	Source
ChatClient	14	1	FeatureIDE
Car	16	6	[KAT16b]
FeatureIDE	19	4	FeatureIDE
FameDB2	21	1	FeatureIDE
Elevator-FeatureModeling	21	3	[MTS+17]
FameDB	22	0	FeatureIDE
APL	23	2	[TBD06]
SafeBali	24	0	FeatureIDE
TightVNC	28	3	FeatureIDE
GPLmedium	38	15	[Lut13]
SortingLine	39	11	[Ana16]
PPU	52	15	[Ana16]
BerkeleyDB	76	20	[KAB07]
axTLS	96	14	[BSL+12]
Violet	101	27	FeatureIDE
uClibc	313	56	[BSL+12]
E-Shop	326	21	[BMB+11]
WaterlooGenerated	580	61	FeatureIDE
Busybox_1.18.0	854	123	[BSL+12]

Table 7.1: All of our evaluated small independent systems and their number of features (#Features), number of constraints (#Constraints), and their origin

Evaluated Systems

We collected a broad set of diverse systems for our evaluation. We grouped our systems into different packages. We created packages based on systems representing the evolution of a product line and named them *history*-based packages. All other systems are named *independent*-based systems, and we grouped them according to the number of their features. Our data packages are publicly available on GitHub[3]. In the following, we give an overview of the packages, their included models, and their source.

Test Packages: Car & GPL

The two first packages contain only small test models that should be used to verify the correct execution when adding a new algorithm to our framework.

The first test package is named `Test_Car` contains only a simplified product line of a car. With 17 features and six constraints, it is one of the smallest of our systems. It was used by Kowal et al. [KAT16b] as an example to explain feature model defects and is since their publication available in FeatureIDE. As the package only consists of one model, it is in the test package for independent-based systems.

The second test package is history-based and consists of the evolution of the graph product line (GPL), a family of classical graph applications. We cover nine time-stamps of GPL with features ranging between 11-23 and constraints between 1-5.

[3]https://github.com/Subaro/Masterthesis-Data

https://doi.org/10.24355/dbbs.084-202009211318-0

Feature Model Name	#Features	#Constraints	Source
Automotive01	2513	2833	FeatureIDE
Linux_2.6.33.3	6467	3545	[BSL+12]
Automotive02_V1	14010	666	FeatureIDE
Automotive02_V2	17742	914	FeatureIDE
Automotive02_V3	18434	1300	FeatureIDE
Automotive02_V4	18616	1369	FeatureIDE

Table 7.2: All of our evaluated large independent systems and their number of features (#Features), number of constraints (#Constraints), and their origin

As it is small, it proves as a test package and also helps us to cover the sampling stability calculation. The GPL models are publicly available on GitHub[4].

Packages for Independent Systems

We constructed three data sets for independent systems. The first package named *Challenge_Small* contains models that we categorized as small models. Table 7.1 shows all models of the small packages along with their number of features, number of constraints, and their origin. All models of these systems are available as examples in FeatureIDE. Whenever it was possible, we provided the original publication (source) that introduces the system.

The second independent data package is named *Challenge_Medium* and contains 116 real-world feature models. The number of features ranges from 1178 to 1408 features and 816 to 956 cross-tree constraints. In 2013, Berger et al. [BSL+13] analyzed the two variability languages *Kconfig* and *CDL* on all available open-source models they could find for both languages. The 116 models used in our package are the *CDL*-based model used in the analysis. They are real-world models of the eCos[5] system, i.e., a highly configurable embedded operating system widely used in multimedia, networking, auto-motive, and even satellite and space-based devices [BSL+13]. Each model represents a different hardware architecture that was available in the 3.0 version of eCos. Knüppel et al. [KTM+18] automatically transformed all 116 models from their original format *CDL* into the FeatureIDE format, making them available for us. For more information about eCos, *CDL*, and the model, we refer the reader to Berger et al. [BSL+13].

The third independent package named *Challenge_Large* contains large-scale real-world feature models. All models are publicly available in FeatureIDE, and therefore, we could extract them in FeatureIDE's XML format. In Table 7.2, we show all of our large models and their information. The automotive models were obtained by a collaboration between FeatureIDE and their industrial partner of the automotive domain. The four *Automotive02* models are monthly snapshots of the same product line. With more than 18000 features, *Automotive02_V4* is the biggest model that is publicly available for us. *Automotive02* is often used in evaluations,

[4]https://github.com/PettTo/Feature-Model-History-GPL
[5]https://www.ecoscentric.com/index.shtml

https://doi.org/10.24355/dbbs.084-202009211318-0

History_FinancialServices		
Date	**#Features**	**#Constraints**
2017-05-22	557	1001
2017-09-28	704	1136
2017-10-20	712	1142
2017-11-20	711	1148
2017-12-22	716	1148
2018-01-23	712	1028
2018-02-20	759	1034
2018-03-26	771	1080
2018-04-23	774	1079
2018-05-09	771	1080

History_FinancialServices_Private		
Date	**#Features**	**#Constraints**
2018-06-26	771	1075
2018-07-03	815	1088
2018-08-15	823	1089
2018-09-24	823	1088
2018-10-15	785	996
2018-11-27	787	996
2018-12-14	786	988
2019-01-24	790	991
2019-02-01	790	991
2019-03-19	795	997

Table 7.3: Provides an overview about all *FinancialServices* models including their timestamp (*yyyy-mm-dd*), number of features (#Features), and number of constraints (#Constraints). The left table contains all publicly available models while the right tables contains our private models

as it was developed for industrial purposes and therefore reflects the requirements of the industry [NMS+18, STS20, Pet18b, KTS+19]. The last model, *Linux_2.6.33.3*, is one of the *Kconfig* models used by Berger et al. [BSL+13] in their comparison of *Kconfig* and *CDL*. *Kconfig*, as a variability language, is used to specify the build-time configurations for the Linux kernel since 2002. Both *Linux_2.6.33.3* and the *Automotive02* models were also part of the scalability challenge of product sampling for software product lines [PTR+19].

Packages for History-Based Systems

The data package *History_Monthly_BusyBox_2007_2010* contains feature models that describe the evolution of the BusyBox system. BusyBox[6] is an open-source, highly configurable system that combines tiny versions of many standard UNIX utilities into a single small executable [Bus]. A publicly available repository[7] contains the feature models for the BusyBox evolution from the 2007-05-20 to the 2010-05-02. We extract 37 monthly snapshots from all models and group them as our data package. The initial model grows from 439 features and 463 constraints to 631 features and 681 constraints. All information on each model can be found in Table A.1.

The second history-based data package named *History_FinancialServices* consists of ten feature models representing the evolution of a system from the financial services domain. The feature models *FinancialServices* were obtained by a collaboration project between FeatureIDE and their industrial partner in the financial services domain. Nieke et al. [NMS+18] published the models for a case study to analyze anomalies for feature model evolutions. Since then, they are also available in FeatureIDE. In cooperation with FeatureIDE, we also gained another ten models of the financial services product line that are currently not published, and we aim

[6] https://busybox.net

[7] https://github.com/PettTo/Measuring-Stability-of-Configuration-Sampling/tree/master/DataAnalysis/data/busybox/models

https://doi.org/10.24355/dbbs.084-202009211318-0

Data Package	timeout	systemIteration	calculateStability
Challenge_Small	10 min	10	false
Challenge_CDL	10 min	5	false
Challenge_Large	24 h	1	false
BusyBox	10 min	10	true
History_FinancialServices	4 h	5	true
History_FinancialServices_Private	4 h	5	true

Table 7.4: Overview about the experiment configurations for the different data packages

to use them as private test data. We refer to the private set of financial services models as *History_FinancialServices_Private*. In Table 7.3, we overview all twenty models and their date when the model was extracted from the product line, their number of features, and their number of constraints.

7.3 Experiments

Now, we introduce the different experiments that we perform for our evaluation. We split them into experiments responsible for computing samples and experiments responsible for computing scores.

7.3.1 Experiments for the Sample Computation

We perform one experiment for each data package. The process for each experiment is the same. We store the models of the current data package inside the input folder for our sampling framework. Then, utilizing our sampling framework, we compute pairwise samples and all evaluation criteria. For that, we set up our sampling framework with the following general configurations: `seed=100`, `maxAlloc=Xmx16g`, `maxAlloc=Xmx4g`, `t=2`. We restrict our evaluation to pairwise product sampling (t=2) as IncLing is a pairwise sampling algorithm. Also, product sampling is very time consuming and coupled with a large amount of data, a small available number of machines, and the duration of our thesis has led us to the decision to restrict the scope of our evaluation to pairwise product sampling. This decision does not mean that our framework can not perform t-sampling for higher t-values, only that more time is needed for this.

Besides the configuration options that are the same for all data-packages, we alter some options for specific packages. In Table 7.4, we show all individual configuration options for each data package experiment. We set the timeout to 10 minutes for the data packages *Challenge_Small*, *Challenge_CDL*, and *BusyBox* as they do not contain large-scale models. As for the *Financial Services* packages, we set the timeout to four hours and for the large-scale models to one day.

The column `systemIteration` indicates how often we repeat the sampling process to erase measuring errors. We calculate most data packages multiple times except for the large-scale models. We estimate that most algorithms will not finish the sampling process for these large-scale models. Therefore, we can assume a runtime

https://doi.org/10.24355/dbbs.084-202009211318-0

of about six days for all large-scale models multiplied by all algorithms. Doing the calculations several times is too time-consuming and would go beyond the scope of our work.

As we are also interested in sample similarity, we set the option `calculateStability` to `true` for the *BusyBox* and *Financial Services* packages. At the end of our experiment, we expect the following results: 1) the computed samples, and 2) the evaluation criteria for all models to compare the sampling algorithms.

To summarize, we use our sampling framework to measure all evaluation criteria for the sampling algorithms Chvatal, ICPL, IncLing, and YASA on the following data packages: *Challenge_Small*, *Challenge_CDL*, *Challenge_Large*, *BusyBox*, *Financial Services*, and *Financial Services_Private*.

7.3.2 Experiments for the Score Computation

We aim to show that our score computation produces precise recommendations. We require a ground truth to verify whether the recommendations computed by our approaches are precise. We use the comparison data generated by the experiments that compute the samples with our framework as our ground truth. Based on the ground truth data, we can decide with our expert knowledge and experience for different use-cases which of our algorithms are more precise, i.e., better suited for the use-case. We split the experiment for the score computation into two sub experiments.

Single Objective

In the first experiment, we aim to find the best algorithm when considering only one objective, e.g., the fastest sampling algorithm. To do so, we compute *NBS*, *SRBS*, *WRBS*, and *IWRBS* based on the data of our ground truth. We restrict the input prioritization to only one criterion (e.g., `[S:1, T:0, C:0, Sim:0, M:0]`) as we are only interested in a single objective. We perform tests based on our ground truth to verify whether the recommendations are precise.

Multiple Objectives

Use-cases considering only one objective are insufficient as real-life situations often require multiple objectives [HPP+13]. For this experiment, we aim to determine which score computation produces the most precise recommendations while considering multiple requirements. To do so, we compute *NBS*, *SRBS*, *WRBS*, and *IWRBS* based on the data of our ground truth for several self-defined use-cases. A use case describes the objectives of the user. For instance, the user wants to find a sampling algorithm that computes a small sample size while consuming the least memory. This use-case can be translated into the following prioritization: `[S:5,T:1,C:1,Sim:1,M:5]`. There are no available use-cases for finding appropriate sampling algorithms that suit our requirements. Therefore, we formulate a set of use-cases that we evaluate in this experiment. In Table 7.5, we show a table containing all our formulated use-cases. After computing all scores, we verify manually whether the computed scores represent the best algorithm based on our ground truth.

https://doi.org/10.24355/dbbs.084-202009211318-0

Prioritization					Use-Case Description
S	T	C	Sim	M	*The user wants to find a sampling algorithm that...*
1	1	1	1	1	weights all criteria equally.
2	2	1	0	0	calculates small samples quickly without considering memory and similarity.
0	1	3	0	0	calculates samples quickly with a high coverage disregarding size, memory, similarity.
3	1	1	0	5	calculates a small sample size while consuming less memory.
1	2	1	5	1	calculates strongly similar samples quickly
2	1	1	-5	1	calculates small strongly dissimilar samples.

Table 7.5: Contains a set of formulated use-cases and their respective transformation into a prioritization. All use-cases are subject of our evaluation.

7.4 Comparing Sampling Algorithms

In this section, we present the results for our first experiment regarding the computation of product samples using our sampling framework and aim to answer some of our research questions. We store all computed samples, criteria data, and generated plots in a GitHub repository[8], which is publicly available. In the course of this chapter, we deal with the research questions $\mathbf{RQ_1}$ to $\mathbf{RQ_5}$. We start with a short introduction to the actual question, followed by an objective observation of relevant results. We then discuss and interpret the results in relation to the research question and try to find an answer to it.

7.4.1 RQ_1 - Which t-wise sampling algorithm calculates the sample fastest?

Product sampling is time-consuming, and thus, the runtime of the sampling process is an essential criterion when choosing an algorithm. It is also one of the most comfortable criteria to measure and is used in most evaluations. Based on the calculation with our data and algorithms, we aim to determine which t-wise sampling algorithm computes the samples in the least possible amount of time.

Observation

In Figure 7.1, we present the data for the sampling time of multiple data packages. For brevity, we skip the data of some packages as they contain similar results. For completeness, we show the data of the skipped packages in Figure A.2. Each of our scatter plots shows the sampling time in the logarithmic scale on the y-axis and the number of features on the x-axis. We can observe that YASA outperforms all other algorithms except for smaller models, e.g., for the BusyBox package. Next, ICPL and IncLing seem to perform quite similar while IngLing performs a notch better when considering all systems' data. The slowest algorithm for all data packages is Chvatal. Especially for the medium and large-scaled systems, Chvatal achieves a sampling time no less than ten times the amount needed of the next best algorithm.

[8] https://github.com/Subaro/Masterthesis-Data

https://doi.org/10.24355/dbbs.084-202009211318-0

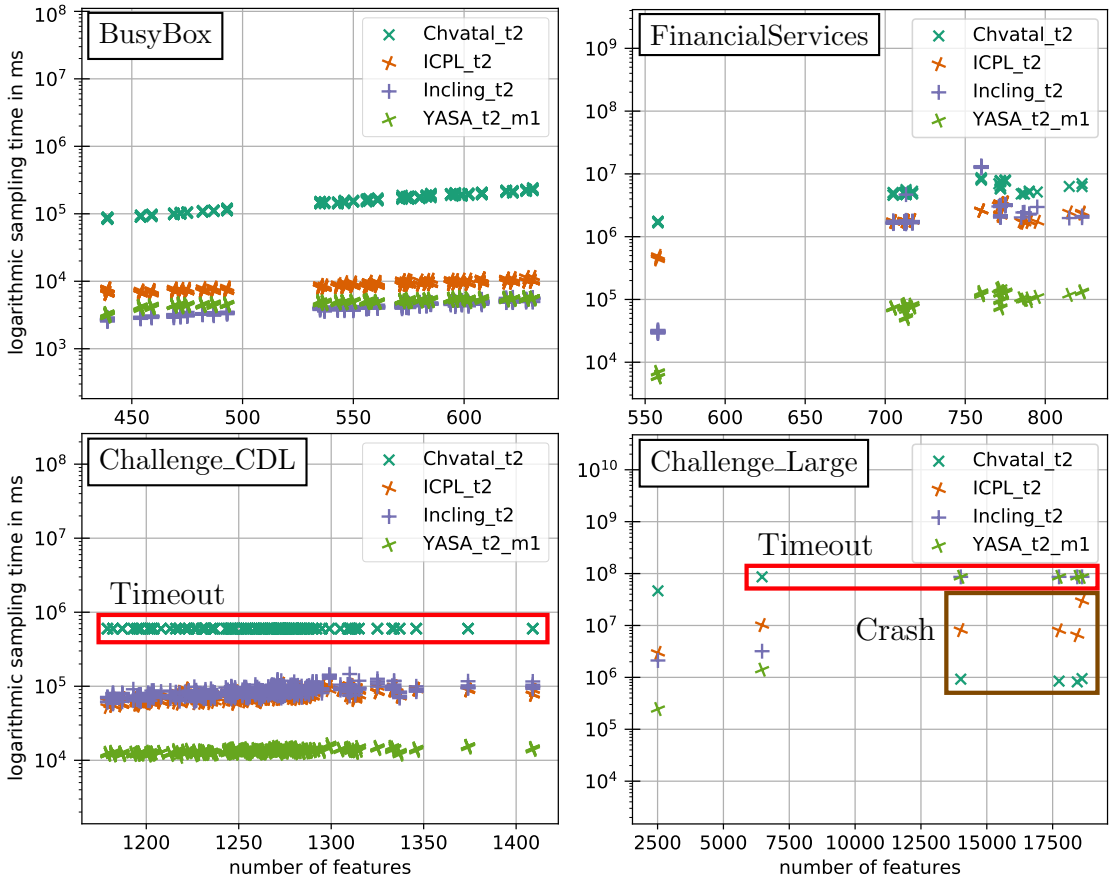


Figure 7.1: Evaluation results of the sampling time required to compute a sample for various data packages (cf. see plot title). Calculations that exceeded the defined timeout or crashed due to errors are especially marked

Two packages only exceeded the timeout for the calculation of samples. First, none of the BusyBox models could be computed by Chavatal in time. Second, none of the sampling algorithms could compute a valid sample for any model with more than 10,000 features. YASA and IncLing exceeded the timeout while Chavtal and ICPL crashed as their process run out of memory. Chavtal also exceeds the timeout for the *Linux_2.6.33.3* model of the large-scale package.

Discussion

The results show us that Chvatal is the slowest, and YASA is the fastest of our algorithms. Chvatal computes the samples on average about 100 times slower than YASA and about ten times slower than IncLing or ICPL. What us surprises is that runtime of IncLing is completely different for each system as IncLing performs well on the *BusyBox* models, mediocre for the *Challenge_CDL* models, and bad for the *FinancialServices* models. The same can be observed for the large models. IncLing performs mediocre for *Automotive01* and a lot better for the *Linux_2.6.33.3*. However, we can find no answer to that behavior. ICPL is an extension of Chvatal, and the improvement of the algorithm is clear to see. On all occasions, ICPL outperforms Chvatal. With all that in mind, we conclude that YASA computes the samples fastest. The algorithm was only outperformed in

https://doi.org/10.24355/dbbs.084-202009211318-0

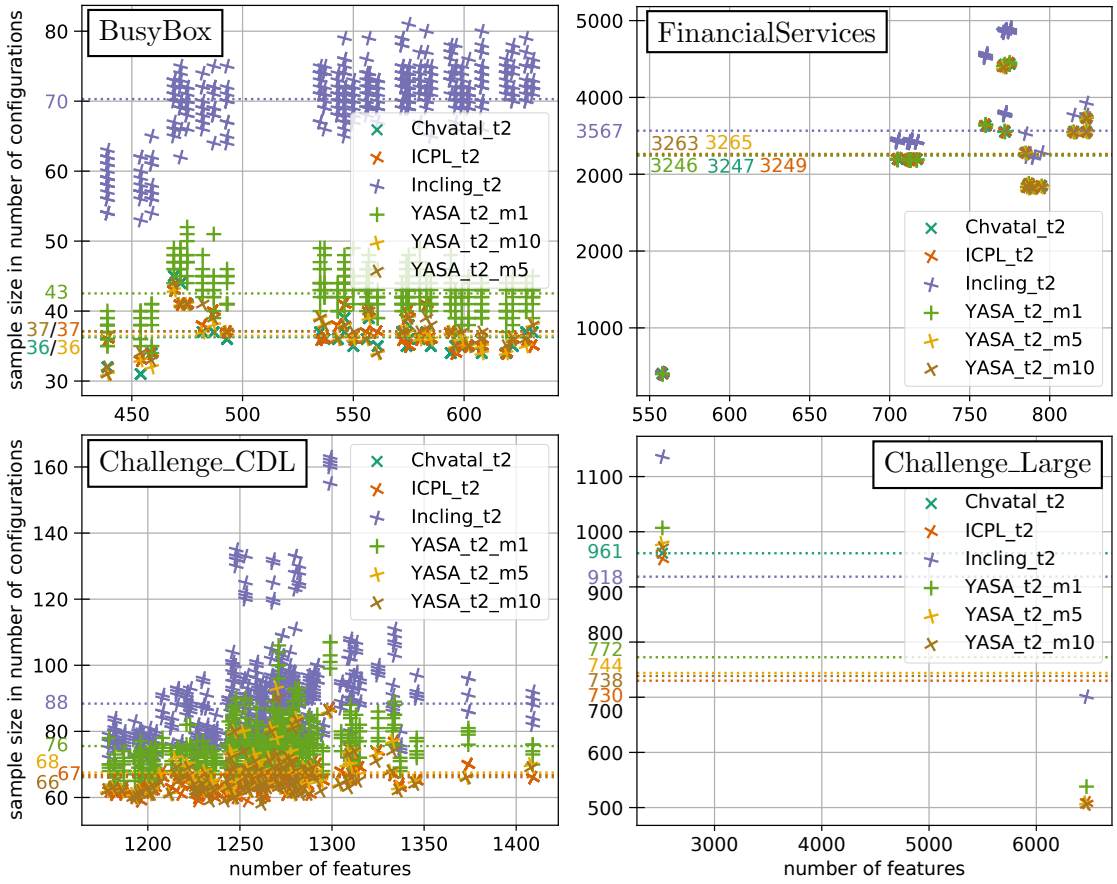


Figure 7.2: Evaluation results of the computed sample sizes for various data packages (cf. see plot title). Some points are not shown because when process timed out or crashed due to errors, and thus, we could not determine the sample size

some smaller systems, but the difference for such small systems is negligible. On the other hand, testing large systems is more critical as the required time for samples seems to grow exponentially, and YASA computes faster for larger systems by a large margin than the other algorithms.

7.4.2 RQ_2 - Which t-wise sampling algorithm computes the smallest samples?

The size of the calculated sample dictates the amount of testing that needs to be done as a larger sample means testing more configurations. With our data at hand, we are interested in determining the sampling algorithm that produces the smallest samples.

Observation

In Figure 7.2, we present the size of the samples that we calculated in our first experiment. For brevity, we skip the data of some packages as they contain similar results. For completeness, we show the data of the skipped packages in Figure A.4. Each of our scatter plots shows the sample size as the number of configurations in

https://doi.org/10.24355/dbbs.084-202009211318-0

the logarithmic scale on the y-axis and the number of features from the sampled system on the x-axis.

We observe that IncLing produces the largest samples for all of our data, having about 5% to 15% more configurations on average. For the *BusyBox* system, the gap is the highest, with about 50% to 100% more configurations. Also, the different samples calculated by IncLing for the same model are often different. The same behavior applies to YASA, while the samples calculated by Chvatal and ICPL are more stable. YASA generally computes a bit larger samples than ICPL and Chvatal. Chvatal and ICPL compute the smallest samples. An exception is the *Financial Services* system's result as the samples calculated by Chvatal, ICPL, and YASA have almost identical sizes.

Discussion

We are surprised in the *Financial Services* results as it requires many configurations to cover all pairwise feature interactions. A reason for that could be the number of complex constraints that reduce the overall number of feature interactions that can be covered by each configuration, resulting in large sample sizes. Another interesting point for *Financial Services* is that Chvatal, ICPL, and YASA compute samples with almost identical sizes with a difference of at most ten configurations.

We think that the derivation of the IncLing and YASA samples for the same model (cf. results for *BusyBox* or *Challenge_CDL*) could originate as we randomize the feature models propositional representation before forwarding them to the sampling algorithms. The internal process of calculating samples for feature models consists of translating the model into a propositional formula in conjunctive normal form (CNF) [ABKS13]. Then, our framework randomizes the order of the clauses, which does not change the expression of the CNF. However, the different order of the clauses makes a difference to the sampling algorithms and results in different samples. Our results for IncLing and YASA could motivate a further investigation of the clause order to improve their process. We identified that Chavatal and ICPL also show that behavior, but their derivation of the samples is negligible.

Another surprise is that Al-Hajjaji et al. [AHKT+16] report that IncLing has only a negligible impact in sample size than existing techniques such as Chvatal and ICPL. However, our results clearly show that Incling consistently calculates larger samples than the other two algorithms. The only reason we could think about is that the order of the clauses is important for IncLing, and randomizing the model before the process makes it harder to compute a sample.

YASA is a sampling algorithm that can be configured to focus on computing the samples fast or try to compute smaller samples for the sake of a longer runtime. We can influence the behavior with the parameter `m` that is passed to YASA. A higher value indicates that more time should be invested to compute smaller samples. Krieter et al.[KTS+20] used the values 1, 5, and 10 in YASA's evaluation. Therefore, we decided to compute our data packages also with the values 5 and 10 for the `m` parameter. With `m5` and `m10`, our results show that YASA computes smaller samples for the *BusyBox*, *Challenge_CDL*, and *Challenge_Large* systems. However, the exact

https://doi.org/10.24355/dbbs.084-202009211318-0

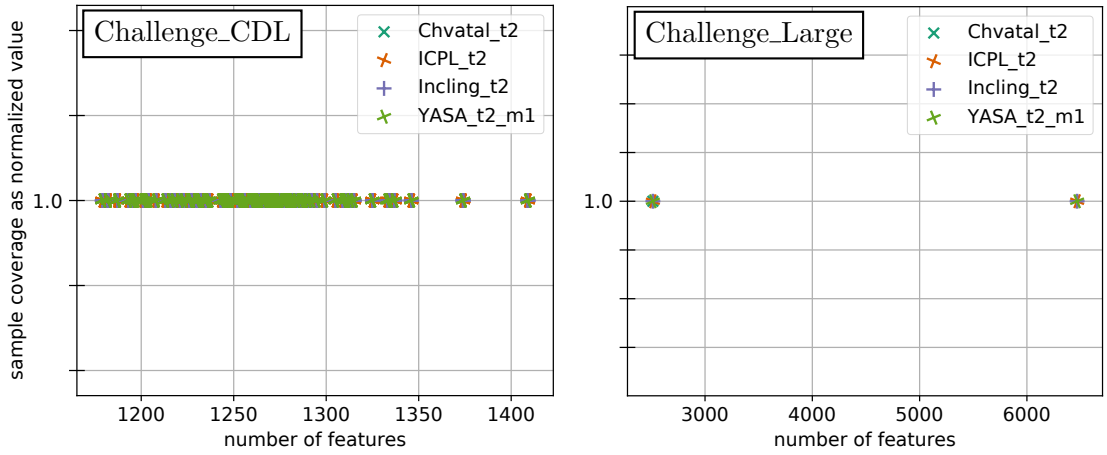


Figure 7.3: Evaluation results of the achieved sample coverage for various data packages (cf. see plot title)

opposite happens for the *FinancialServices* models as the computed samples are larger by about 20 configurations.

CHVATAL performed poorly on average for the *Challenge_Large* system. However, the average is only bad as the computation for the second model did not finish, which generally require smaller samples.

Based on all our results, we conclude that YASA and ICPL compute the smallest sample sizes. CHVATAL also computed quite small samples. However, CHVATAL did not consistently compute samples and fails for the larger systems. Also, we think that ICPL generally smaller samples than YASA, but when increasing the `m` parameter, the difference between the two algorithms become negligible.

7.4.3 RQ_3 - Which t-wise sampling algorithm achieves the highest t-wise coverage?

The coverage shows how many t-wise feature interactions are covered. When testing, it is wanted to achieve complete coverage while areas of application such as the statistical computation of product lines do not require complete coverage. As we are interested in t-wise sampling for testing purposes, we aim to determine the sampling algorithm that achieves the highest t-wise coverage.

Observation

In Figure 7.3, we present the achieved coverage as nominal values of the samples that we calculated in our first experiment. For brevity, we skip the data of some packages as they contain similar results. For completeness, we show the data of the skipped packages in Figure A.4. Each of our scatter plots shows the achieved coverage as nominal values (i.e., 0 means 0% and 1 means 100%) on the y-axis and the number of features from the sampled system on the x-axis. All computed samples achieve complete coverage.

https://doi.org/10.24355/dbbs.084-202009211318-0

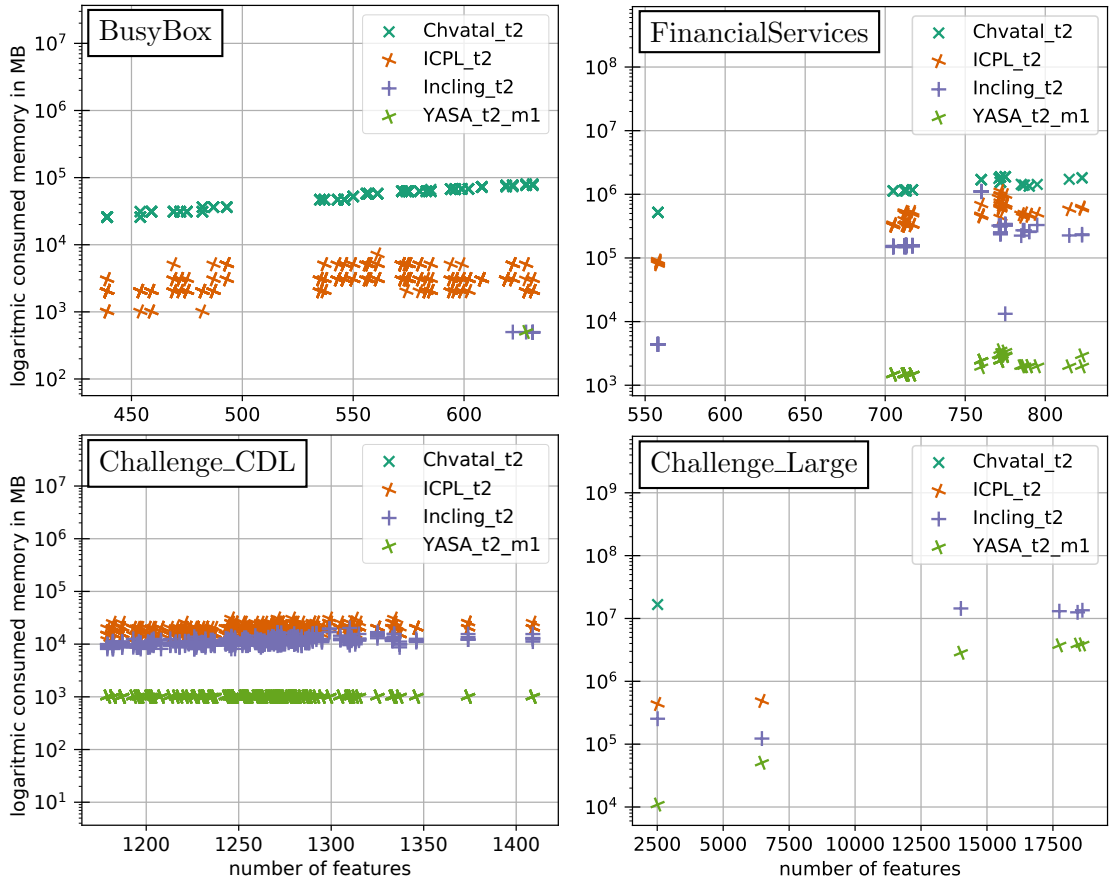


Figure 7.4: Evaluation results of the consumed memory for the sample process for various data packages (cf. see plot title)

Discussion

All of our evaluated sampling algorithms guarantee complete coverage. Our framework verified this statement as each sample achieved complete coverage. Therefore, we conclude that for the aim of computing a high/complete coverage, any of our evaluated algorithms are suitable.

7.4.4 RQ_4 - Which t-wise sampling algorithm consumes the least memory in the process?

Memory consumption is often not measured in evaluations for sampling algorithm. However, we think with the trend towards resource optimized environments. We need to identify sampling algorithms that can be executed in such restricted environments. As our sampling framework measures the memory consumption of evaluated sampling algorithms, we aim to find the algorithm that produces the least possible amount of memory.

Observation

In Figure 7.4, we present the evaluation results for the consumed memory of each algorithm. For brevity, we skip the data of some packages as they contain similar

https://doi.org/10.24355/dbbs.084-202009211318-0

results. For completeness, we show the data of the skipped packages in Figure A.3. Each of our scatter plots shows the consumed memory as MB in the logarithmic scale on the y-axis and the number of features from the sampled system on the x-axis. Consumed memory expresses the memory that is allocated for the sampling process and then later freed.

The first point to notice is that some data points are missing. This behavior occurs as we extract the consumed memory from the garbage collector file created by the sampling process. However, when the process finishes too fast, no data can be extracted, and thus, some points are missing. This only happens for sampling processes that need less than four to seven seconds to finish. Another reason is that the sampling processes timed out or crashed. Then, we also could not extract memory information.

CHVATAL has the highest memory consumption and consumes about 10 to 60 times the memory required by the next best algorithm ICPL. ICPL consumes consistently more memory than INCLING. The least amount of memory consumption is measured for YASA.

Discussion

YASA consumed by far less memory than the other sampling algorithms. This could have two reasons. First, the approach from YASA is simply more efficient, and thus, also needs less memory. Second, YASA is newer and could use more efficient and modern data structures for their computation, which were not available for the other algorithms' implementations. Based on our results YASA should be used if memory consumption is an important criterion.

7.4.5 RQ_5 - Which t-wise sampling algorithm calculates the most similar and which the most dissimilar samples?

The similarity value of two samples can be a good or bad quality for a testing strategy depending on the user [HB10, Pet18a]. There are two use cases for similarity. First, use-case `A`, the user wants to find bugs in system functionalities they have already seen. Thus, a high similarity value is good because our sample after an evolution is very similar to the previous one. Second, use-case `B`, the user wants to find bugs in system functionalities that he has not seen before. Thus, a low similarity value is good because more different configurations are tested than the previous sample. The use case depends on the user. We aim to determine the sampling algorithm that computes the most similar samples for use-case `A` and the most dissimilar samples for use-case `B`.

Observation

In Figure 7.5, we present the evaluation results for the sample stability of our history-based systems. Each of our scatter plots shows the sample stability between the samples of two timely connected models as normalized values on the y-axis and the number of features from the sampled system on the x-axis. We could only compute the sample stability for our history-based systems, and thus, we only provide plots

https://doi.org/10.24355/dbbs.084-202009211318-0

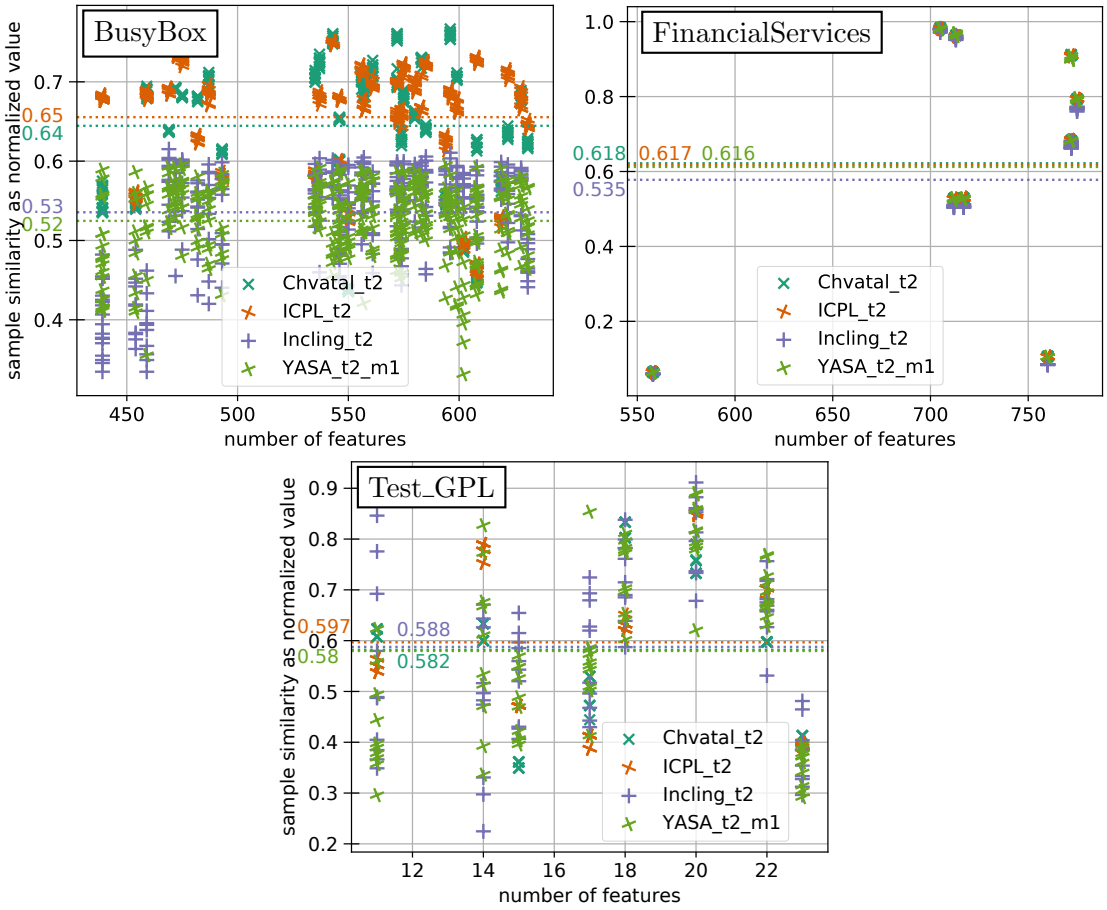


Figure 7.5: Evaluation results of the similarity between computed samples for history-based data packages (cf. see plot title)

for *TestPackage_GPL*, *BusyBox*, and *FinancialServices*. We also introduce lines that show the average similarity value for each algorithm as the results are mixed and hard to compare. We also colored the value text accordingly whenever an average value cannot be referenced to a line.

We can observe that ICPL computes the most similar samples for *BusyBox* and *GPL*. For *FinancialServices*, ICPL is also one of the algorithms that compute the most similar samples. CHVATAL computes the most similar samples for *FinancialServices* and almost as similar samples as ICPL for the other systems.

Both INCLING and YASA compute the most dissimilar samples. YASA samples for *BusyBox* and *GPL* are about 1% to 2% more dissimilar than the samples calculated by INCLING. However, INCLINGs samples for the *FinancialServices* system are about 10% more dissimilar than YASAs.

Discussion

The results for *FinancialServices* surprise us as the samples of YASA, CHVATAL, and INCLING show almost the same similarity. We think that the system somehow restricts the selection of possible configurations in such a way that a large number of configurations are always required for a sample. Our results for the sample size

https://doi.org/10.24355/dbbs.084-202009211318-0

already showed that the samples for the *FinancialServices* system contain about 3,000 to 5,000 features. We think that the large number of necessary configurations for the sample results in all algorithm computing samples with the same similarity.

The similarity average of each algorithm changes too often with each system. For example, YASA computes the most dissimilar samples for *BusyBox* and computes one of the most similar samples for the FinancialService. It seems that the sample similarity depends on the system.

Based on our results, we conclude that ICPL computes the most similar samples as the algorithm reliably produces similar samples. Both IncLing and YASA compute dissimilar samples for *BusyBox* and *GPL*. YASAs samples are about 1% to 2% more dissimilar on average than the samples computed by IncLing. However, IncLings samples for the *FinancialServices* system are about 10% more dissimilar than the samples of YASA, making the previous difference negligible. Hence, we conclude that IncLing computes the most dissimilar samples for our systems.

7.5 Ranking Sampling Algorithms by Score

In this section, we present the results for our second group of experiments regarding the computation of scores using our approaches and aim to answer some of our research questions. We store all computed scores and generated plots in a GitHub repository[9], which is publicly available. In the course of this chapter, we deal with the research questions $\mathbf{RQ_6}$ to $\mathbf{RQ_7}$. We start with a short introduction to the actual question, followed by an objective observation of relevant results. We then discuss and interpret the results in relation to the research question and answer it.

7.5.1 RQ_6 - Which score computation computes the correct algorithm for only one requirement?

All of our score approaches follow the same idea to create a subscore for each evaluation criteria. Now, we aim to verify that these subscores are correct. For that, we compute the scores for all approaches by restricting the input prioritization to only one value. For example, to verify whether the approaches compute the correct algorithm regarding the sampling size, we set the input prioritization to `[S:1,T:0,C:0,Sim:0,M:0]`.

Observation

In Figure 7.6, we present the evaluation results for the score calculation with one requirement. Each column of plots shows the scores for a specific approach (e.g., NBS) based on our evaluated sampling algorithm's available data. Each row of plots shows the prioritization that was used to calculate the score. We assume that only one requirement is set to 1, while all others are set to 0. For instance, the first row of plots shows all scores that used the prioritization [S:0, T:1, C:0, Sim:0, M:0]. The right text (e.g., Time=1) indicates which requirement was used for the computation. Each of our bar plots contains four bars, one for each of our evaluated algorithms

[9] https://github.com/Subaro/Masterthesis-Data

https://doi.org/10.24355/dbbs.084-202009211318-0

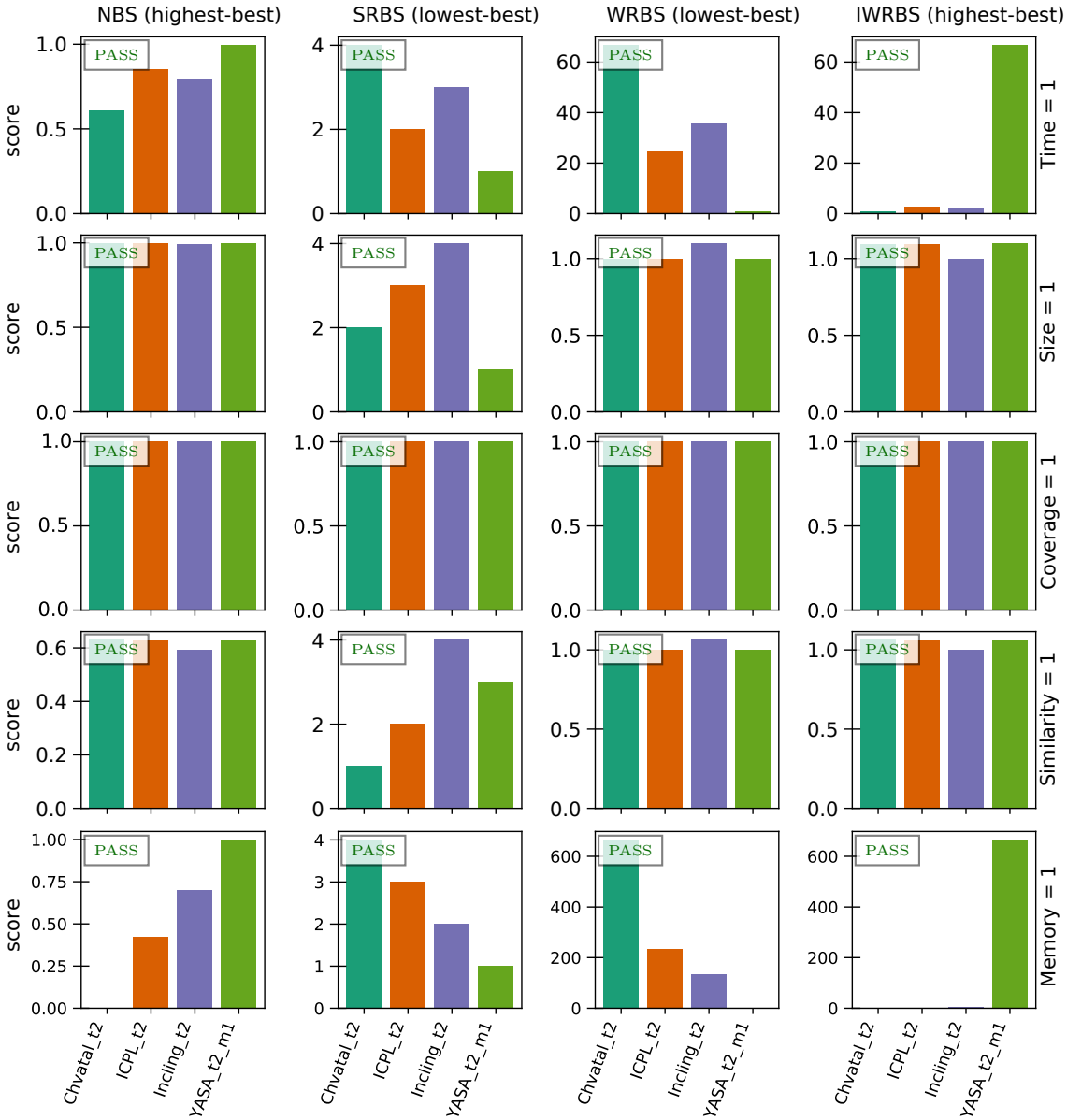


Figure 7.6: Evaluation results for the score calculation with one requirement. Each column of plots shows the scores for a specific approach (e.g., NBS) based on the available data of our evaluated sampling algorithm.

from the first experiment, and the y-axis shows the value of the calculated score. For brevity, we only show the results for the *FinancialServices* systems, and the remaining plots can be found in our repository[10]. We perform verification tests to check whether the best score represents the best algorithm. The verification test is marked for each plot at the top left, showing *Pass* when the best score is the best algorithm for our requirement.

We process the results for each row iteratively. We begin with the first row (i.e., Time=1) that uses the prioritization [S:0, T:1, C:0, Sim:0, M:0] for their computation. We can see that the ranking is for all the same with CHVATAL in the last place,

[10]https://github.com/Subaro/Masterthesis-Data

https://doi.org/10.24355/dbbs.084-202009211318-0

IncLing in third place, ICPL in second place, and YASA with the best results. We used the ground truth data for our verification testing. Our verification testing showed that the ranking is precise for the prioritization, and thus, we mark all as passed.

The second row (i.e., Size = 1) only considers the sample sizes to calculate the scores. The computed scores are not as distinctive as the results for the sampling time. Notably, the NBS shows almost the same values. Still, the actual ranking of the algorithms is the same for all scores. The SRBS can show the differences between the algorithms more clearly. In terms of sample size, all scores indicate that YASA performs best, then Chvatal, ICPL, and IncLing. With our ground truth, we could verify the results and mark all approaches as passed.

The third row (i.e., Coverage = 1) aims to compute a score using only the coverage as evaluation criteria. All sampling algorithms guarantee complete coverage. So our approaches rank the algorithms all as equals. All verification test pass.

The fourth row (i.e., Similarity= 1) ranks algorithms based on their computed samples' similarity. The scores for Chvatal, ICPL, and YASA seem identical for all score approaches except NBS. However, our textual representation shows us that they are not identical. Based on our values, we can see that all approaches compute the same recommendations: 1) Chvatal, 2) ICPL, 3) YASA, and 4) IncLing. We checked all values with our verification tests, and they all pass.

The last row (i.e., Memory = 1) only considers sampling process's consumed memory to calculate the scores. We see that all scores compute the same recommendations: 1) YASA, 2) IncLing, 3) ICPL, and 4) Chvatal. We verified the results with our ground truth and marked all results as passed.

Discussion

The scores for NBS are often too hard to distinguish between the evaluated algorithms, especially for the sample size. The sample sizes of Chvatal, ICPL, and YASA are, in fact, almost identical, i.e., sample size between 3246-3249 (cf. Figure 7.2). However, IncLing computed a sample with 3567 configuration, but the different in the NBS scores between the algorithms is negligible. On the other hand, SRBS is too distinctive as it completely omits the relations between the scores of all algorithms and simplifies them into a ranking. Hence, it is not clear whether the two algorithms perform almost identically. WRBS and IWRBS show reasonable results in the ranking and also in the relation between the scores. For example, we can conclude from the WRBS plot and the underlying data for *Size=1* that Chvatal, ICPL, and YASA perform better than IncLing without affecting the correct ranking of the algorithms.

All passed verification tests show that all approaches are reliable to compute the best algorithms when only considering one requirement. Thus, we conclude that all approaches deliver a distinctive result to find the algorithm that performs best for one requirement.

https://doi.org/10.24355/dbbs.084-202009211318-0

7.5.2 RQ_7 - Which score computation computes the most precise recommendation for a given prioritization?

For users, one requirement is often insufficient, and multiple requirements need to be considered to find an appropriate sampling algorithm [HPP+13]. With this question, we aim to determine which approach produces the best recommendations. For that, we performed a set of experiments on a defined set of use-cases.

Data	NBS				SRBS				WRBS				IWRBS			
	CH	**IC**	**IL**	**YA**	**CH**	**IC**	**IL**	**YA**	**CH**	**IC**	**IL**	**YA**	**CH**	**IC**	**IL**	**YA**
1. Use-Case: `[S:1,T:1,C:1,Sim:1,M:1]`, `Sim` is 0 for non history-based systems																
Challenge_Small	3.87	3.94	3.974	**3.977**	11	8	8	**7**	188.8	8	4.95	**4.07**	4.27	57.69	120.5	**188.5**
BusyBox	4.29	**4.64**	4.53	4.52	12	**10**	**10**	12	479	11.41	6.16	**5.63**	6.17	39.57	**149.97**	143.2
FinancialSystems	3.24	3.9	4.08	**4.62**	12	11	14	**7**	731	261.8	169.6	**5**	5.16	8.67	9.89	**731**.2
Challenge_CDL	-	3.87	3.85	**3.98**	-	2	3	**1**	-	26.66	19.92	**4.13**	-	11.13	10.75	**66.1**
Challenge_Large	2.88	3.2	8.92	**23.8**	8	**5**	12	19	40.19	**4.52**	27.75	13.1	6.48	**43.72**	4.85	10.17
2. Use-Case: `[S:2,T:2,C:1,Sim:0,M:0]`																
Challenge_Small	4.795	4.925	4.948	**4.951**	13	9	**11**	**11**	39.31	8	5.54	**5.15**	5.53	18.06	**39.31**	38.67
BusyBox	4.48	4.97	**4.99**	4.98	**11**	**11**	**11**	**11**	79.99	7.3	6.87	**5.74**	6.88	40.98	**79.99**	68.71
FinancialSystems	4.21	4.7	4.58	**4.98**	13	11	15	**5**	129.1	49.96	69.33	**5**	5.2	8.57	6.81	**129.29**
Challenge_CDL	-	4.74	4.72	**4.96**	-	**7**	13	**7**	-	14.52	16.39	**5.26**	-	19.27	17.14	**93.42**
Challenge_Large	4.77	4.75	4.94	**4.98**	**7**	**7**	17	13	7.17	**6.04**	15.55	14.83	9.97	**14.24**	5	5.4
3. Use-Case: `[S:0,T:1,C:3,Sim:0,M:0]`																
Challenge_Small	3.91	3.98	**3.991**	3.99	7	6	**4**	5	21.15	5.5	**4**	4.03	4	10.26	**21.15**	20.62
BusyBox	3.74	3.99	**3.99**	3.99	7	6	**4**	5	41.5	5.13	**4**	4.2	4	21.1	**41.5**	35.2
FinancialSystems	3.61	3.85	3.8	**3.99**	7	5	6	**4**	66.05	26.48	36.07	**4**	4	5.69	4.91	**66.05**
Challenge_CDL	-	3.87	3.86	**3.98**	-	5	6	**4**	-	8.77	9.37	**4**	-	10.81	10.07	**48.04**
Challenge_Large	3.88	3.87	3.97	**3.99**	5	**4**	7	6	5.08	**4**	8.36	8.31	5.57	**8.36**	4	4.01
4. Use-Case: `[S:3,T:1,C:1,Sim:0,M:5]`																
Challenge_Small	9.71	9.83	9.94	**9.95**	31	22	24	**17**	865.4	24	14.22	**10.16**	10.8	252.8	523.8	**865.5**
BusyBox	9.29	9.96	**9.99**	9.99	28	25	**19**	22	584	37.49	12.81	**10.76**	12.82	111.42	**584.7**	574.90
FinancialSystems	4.6	6.96	8.27	**9.97**	31	27	26	**10**	3391	1204	704.2	**10**	10.3	21.11	30.84	**3392**
Challenge_CDL	-	9.837	9.838	**9.976**	-	21	23	**13**	-	104.2	67.48	**10.39**	-	17.78	19.48	**144**
Challenge_Large	4.88	6.5	**44.6**	11	26	**13**	32	23	186.6	**11.56**	109.5	37.03	14.31	**190.7**	14.27	40.42
5. Use-Case: `[S:1,T:2,C:1,Sim:5,M:1]`																
BusyBox	7.61	**8.24**	7.66	7.60	24	**17**	23	30	192.5	17.54	12.06	**11.82**	12.08	62.67	**192.5**	179.4
FinancialSystems	6.36	7.27	7.25	**8.12**	**20**	21	33	**20**	798.1	289.3	206.9	**10.01**	10.41	15.6	15.8	**798.5**
6. Use-Case: `[S:2,T:1,C:1,Sim:-5,M:1]`																
BusyBox	1.43	1.7	2.32	**2.37**	1	6	-4	**-9**	144.9	6.43	0.76	**-0.7**	0.73	33.67	**144.9**	138.9
FinancialSystems	0.45	1.12	1.52	**1.84**	8	2	-6	**-10**	726	256.8	164.3	**-0.01**	-0.11	3.4	4.89	**725.9**
Precision	16 / 24 (66.6%)				16 / 24 (66.6%)				**24 / 24 (100%)**				18 / 24 (75%)			

Table 7.6: All evaluation results for our defined use cases (cf. Table 7.5). The columns NBS, SRBS, WRBS, and IWRBS show the computed scores based on the data of our evaluated sampling algorithms: *CH*=Chvatal, *IC*=ICPL, *IL*=IncLing, and *YA*=YASA. We marked cells according to their precision. Green cells indicate that the score approach recommends the algorithm we would manually recommend based on our experience and domain knowledge. Red cells shows that the computation recommends another algorithm. In these cases, we marked our recommendation in blue

https://doi.org/10.24355/dbbs.084-202009211318-0

Observation

In Table 7.6, we present the evaluation results for the score calculation for our defined set of use-cases (cf. Table 7.5). The *Data* columns contain the name of the system that is used for the score calculation. The other columns *NBS*, *SRBS*, *WRBS*, and *IWRBS*, contain the respective scores that we calculate based on the ground truth. Further, for each score computation, we compute four scores, one for each sampling algorithm. These algorithms are abbreviated in the table as *CH*=Chvatal, *IC*=ICPL, *IL*=IncLing, and *YA*=YASA. The best values for each computation are marked as bold.

After calculating all scores, we revised all results and marked the cells of the scores with the following colors:

- The score computation recommends the same algorithm that we would manually recommend based on our ground truth, experience, and domain knowledge.
- The score computation recommends a different algorithm that we would manually recommend.
- For the case, that a score cell was marked red, we also mark our recommendation in blue.

In the last row of the table, we count how often a score computation coincides with our manual recommendation. Based on the precision, we can see that WRBS performs best with always recommending the same algorithm that we would recommend. The next best computation is IWRBS, with 75% followed by SRBS and NBS, both with a precision of 66.6%.

Discussion

The NBS reaches a 66.6% precision, and we identified two problems that could be the source of error. When considering the sample size as a criterion, then NBS produces more failures with only a small gap between their recommendation and ours. The problem could be that the transformation of the sample size into the nominal value is too crude and results in having the sample size almost being ignored. If the difference between the algorithms' sample sizes is more distinctive, then NBS could produce more accurate results. The second problem is the nominal value for memory consumption. Sometimes the score computation produces non-nominal values for memory consumption, and thus, negatively impacts the results. For example, the results of the *Challenge_Large* systems for the fourth use-case report an NBS score of 44.6 for IncLing. So we think it is necessary to find better approaches to transform both the sample size and the consumed memory into nominal values to improve NBS further.

The results for SRBS surprise us. The approach completely omits the relation between the algorithm and uses only simple ranks to find a recommendation. In four cases, SRBS recommends multiple algorithms with the best values, including the algorithm we recommend. We did not count these occasions as they also recommend bad algorithms. For instance, the *BusyBox* system results for the second use-case show that SRBS would recommend all four algorithms. However, the Chvatal algorithm performs multiple times worse than any other three algorithms for the system.

https://doi.org/10.24355/dbbs.084-202009211318-0

Based on our results, WRBS recommends the most precise algorithm for all of our defined use-cases. Further, the ranking of the other algorithms seems solid, and skipping the best algorithm and choosing the second-best seems reasonable when their WRBS is similar. The inverse approach IWRBS reached only a precision of 75%, which is interesting. Even more impressive is that most of the failures were computed for the *BusyBox* system. When we introduced the inverse approach, we already identified the problem that IWRBS could produce different recommendations as WRBS. The WRBS would produce recommendations that create more significant gaps between bad algorithms. IWRBS, on the other hand, produces recommendations that have smaller gaps for worse ranked algorithms and produce larger gaps between the best algorithms. Our score computation aims to find the most suitable algorithm, so we conclude that the WRBS produces the most suitable recommendations for our use-cases.

7.6 Threats to Validity

In this section, we explain possible threats to validity for our empirical evaluation. We split the threats into internal and external.

Internal Validity

Translation to Feature Model: Transforming the configurable subject system into a function model may be a source of threats. Many of our evaluated systems were exported from FeatureIDE. We then tried to locate the sources of each system. Since no source could be found for some systems, we can not explain their integrity.

The industrial systems *FinancialServices* and *Automotive* were designed as feature models by the cooperative manufacturers and used for real projects. The feature names have been changed by the obfuscator provided in FeatureIDE. This process does not affect the semantics of the models. Using these models in real projects shows their integrity.

The CDL model we use in our evaluation was translated from CDL and KConfig into the FeatureIDE file format by Knüppel et al. [KTM+18]. The authors follow the mapping concepts between CDL, KConfig, and Feature models. Their first threat to validity is their transformation, as they remove possible simple cross-tree constraints (e.g., constraint references a non-existent feature) that reduce the complexity of the model. Another threat is that Knüppel et al. do not minimize the models by performing any logical optimizations. On the contrary, in rare cases, new abstract feature stress has been added.

Exporting Feature Models: As part of our evaluation, the models in FeatureIDE format are converted to DIMACS format before they are given as input to the evaluating algorithms. This conversion is done by the exporter provided in FeatureIDE [KPK+17, TKB+14]. One threat about our integrity is that we have no further information regarding the transformation from FeatureIDE format into DIMACS. We performed some small tests with our models and identified no changes in the semantics (e.g., number of features) of our models before and after exporting into DIMACS.

https://doi.org/10.24355/dbbs.084-202009211318-0

Randomizing Feature Models: During our evaluation, we randomize our models' logical presentation before giving them input to the algorithms. This process does not change the logical statement. Only the order of the clauses of the logical formula is changed. The randomization process in our experiment always uses a seed, which we have described in the experiment. This seed ensures that all algorithms always receive the identically randomized model.

Sampling Framework: Another threat for our integrity is our sampling framework. We have used the framework to perform measurements for the comparison of sampling algorithms. We have developed and implemented the framework ourselves, trying to minimize the influence of the framework on the measurements. To do this, we start each sampling process as a separate machine and run all experiments multiple times, except for the very large models, to eliminate any measurement errors.

Measuring Memory: To measure memory consumption, we used the garbage collector files of the Java Virtual Machine (JVM). It is possible to connect directly to the JVM via the API and read the memory results. However, many benchmarks use the garbage collectors files as they produce accurate results, can be analyzed separately, and do not require another program in parallel that constantly reads all memory values [BGH+06]. The data we can extract from the GC Log files is less than what a separate program can capture. However, the data of the GC log files is sufficient for our measurement, and therefore, we have decided to use the GC log approach.

JVM Warm-Up: The JVM warm-up phase occurs after a new JVM is started. Besides executing the actual program, Java loads classes, interprets the bytecode, stabilizes the cache, and performs *Just-in-time* (JIT) optimizations [BGH+06]. These processes often impact the measurements negatively. Lion et al. [LCS+16] have shown that the warm-up phase consistently lasts up to 21 seconds and has only a small impact on very time-consuming programs. Therefore, we decided to ignore the warm-up phase for all experiments because product sampling is a time-consuming process. We think only the small systems would show any noticeable differences when performing a warm-up task before starting the actual sampling process.

External Validity

Sampling Algorithms: We have used four sampling algorithms and their implementation in the form of libraries. We have not paid attention to implementation-dependent optimizations that differ from the published algorithms. Also, we did not change any of the algorithms' internal parameters to get more optimized results. The only exception was the YASA algorithm, which allows us to configure the trade-off between sample size and runtime. We have evaluated YASA several times with different trade-offs, as this trade-off is a characteristic feature of YASA.

For the implementations, we have used popular libraries. The library SPLCAT offers an implementation for Chvatal and ICPL and is used in many publications [DPL+15, HPP+14, KTS+20, HMGB16]. For IncLing and YASA, we use the implementation of the popular feature modeling tool FeatureIDE.

https://doi.org/10.24355/dbbs.084-202009211318-0

Comparison of Sampling Algorithms: Another threat to our external validity is that we evaluate only four algorithms. That is only a small subset of all available algorithms, and thus, we cannot say that our results can be applied generally when other algorithms are considered. However, our thesis's task is to transform product sampling into a shared task that requires the community's participation to achieve a complete comparison of all available sampling algorithms.

Homogeneity of Models: In our evaluation, we have used a large number of different systems. The important point is that we selected our systems from several domains and in different sizes to transfer the results to other systems.

7.7 Summary

We started by introducing the research question that we aim to answer in our empirical evaluation. Our research questions cover the comparison of sampling algorithms and the selection of appropriate sampling algorithms based on our score calculation. Then, we described the setup of our empirical evaluation containing the machine specifications, versions of software implementations, and the individual experiments. We split the experiments into two groups. The first group uses the sampling framework to measure the evaluation criteria of the sampling algorithms: Chvatal, ICPL, IncLing, and YASA. The second group of experiments computes our evaluation scores based on the results generated by our sampling framework in the first group of experiments. Afterward, we started to present the results for the comparison of sampling algorithms and systematically answered each research question (cf. Section 7.1). In the last part of our evaluation, we focused on the computation of scores to recommend the most appropriate sampling algorithm for a given prioritization. We investigated all score computation approaches and their correctness for both use-cases focusing on only one requirement and use-cases focusing on a set of requirements (i.e., a prioritization). With our results, we answered the last two research questions regarding the computation of scores.

https://doi.org/10.24355/dbbs.084-202009211318-0

https://doi.org/10.24355/dbbs.084-202009211318-0

8. Related Work

8.1 Reproducibility in Computer Science

Recreating the results of already published papers is essential to prove their credibility. However, the software and data of published papers are often not distributed, making the process tedious or, in the worst case, impossible. Stodden et al. [Sto10] addressed the lack of credibility in computational science caused by the inability of researchers to verify or reproduce published results independently. They propose and describe a set of recommendations for researchers, journals, and funding agencies to improve the overall situation, known as science's reproducibility crisis [PGWS19]. The history of the terminology used to describe the reproducibility for a piece of research contains contradictory definitions and missing understandings, and thus, it is somewhat ill-defined to this day [Ple18]. ACM introduced a terminology for reproducibility, dividing it into three categories: *Repeatability* (i.e., the same team evaluates the experiment with the same setup), *Replicability* (i.e., a different team evaluates the experiment with the same setup), and *Reproducibility* (i.e., a different team evaluates the experiment with a different setup) [Res17]. Potthast et al. [PGWS19] introduce TIRA as a cloud-based evaluation-as-a-service system to facilitate repeatability, replicability, and aspects of reproducibility by providing a platform for participant-in-charge software submissions. Hanbury et al. [HMB+15] provide an overview of the collective evaluation-as-a-service systems. Not many cloud-based evaluation systems such as TIRA are available as the paradigm itself is still new.

Our thesis aims to transform product sampling into a shared task by using our sampling platform. Our concepts and implementations are not dependent on TIRA and can be used independently. For the shared task, we can only initiate the first step by providing our framework and score computations via TIRA, while providing the first data. Our evaluation did not cover all existing sampling algorithms, so it a community effort to compare all sampling algorithms.

https://doi.org/10.24355/dbbs.084-202009211318-0

8.2 Product Sampling

The literature covers many techniques to test product lines [CMMCDA14, TAK+14, vRAK+13]. Sampling, also known as sampling-based analysis, reduces the number of products to test [TAK+14, vRAK+13]. Combinatorial interaction testing (CIT) is a sampling technique to cover feature interaction to a certain degree to ensure that most product failures are detected [CDFP97, JHF12a, POS+12, OMR10, CMMCDA14, PSK+10]. Sampling via CIT is often used to detect errors in products. However, sampling is also used to compute statistics on product lines [OBMS17, OGB+19b, PAP+19, PAMJ20] or to detect products that fulfill a constraint based on non-functional properties[KGS+19, PAMJ20].

In 2018, Varshosaz et al. [VAHT+18] published a survey for product sampling of software product lines. They propose a classification for product sampling techniques based on input data, kind of algorithm, and the achieved coverage. Additionally, they create an overview of existing tool support for the various techniques. Furthermore, they identified 48 papers in the literature and classified them. In our thesis, we extend the survey by 16 new publications and classified them accordingly. Of these works, ten were identified and classified by us. They include a family of fundamental greedy CIT sample generation algorithms [CDFP97], real uniform random sampling for product lines [OBMS17, OGB+19b, PAP+19, PAMJ20, OGB19a], a new configurable t-wise sampling algorithm [KTS+20], distance-based sampling to find optimal product concerning their performance [KGS+19, PAMJ20], and an extensive report of comparing multiple sampling strategies [HNA+19]. The remaining six papers have been added since the survey was published on their website. We have reviewed these classifications and included them in our extension.

8.3 Comparing T-Wise Sampling Algorithms

Whenever an author develops a new sampling algorithm, he naturally want to evaluate it and compare it with current competing algorithms. We present some of these works and try to restrict them to the scope of our thesis (i.e., only algorithms we processed in the evaluation or survey).

Chvatal [Chv79] published an algorithm to generate covering arrays for any strength, i.e., the degree of interactions between sets. The resulting covering arrays always reach a 100% coverage of interactions, but they are not minimal, and thus, the algorithm is a greedy heuristic. Johansen et al. [JHF11] adapted Chvatal's algorithm to generate t-wise samples of feature models. Then, Johansen et al. [JHF12a] developed ICPL based on their adaption of the Chvatal algorithm by detecting and handling invalid feature combinations more efficient. They compare ICPL against the sampling algorithms Chvatal [JHF11], CASA [GCD11], IPOG [LKK+07], and MoSo-PoLiTe [OMR10]. They compare these algorithms regarding the sample sizes and time to generate the samples. On the contrary, we examine several evaluation criteria and compare the algorithms in addition to ICPL, Chvatal, IncLing, and YASA. The results of their work coincide with the result of our evaluation and show that ICPL is faster than Chvatal and has no drawback concerning sample sizes.

https://doi.org/10.24355/dbbs.084-202009211318-0

Cohen et al. [CDFP97] developed a family of fundamental greedy CIT sample generation algorithms that exploit calculations made by modern Boolean satisfiability (SAT) solvers. They performed an evaluation to measure the sampling size and sample sizes for their family of algorithms.

Al-Hajjaji et al. [AHKT+16] introduced the incremental pairwise sampling algorithm IncLing. Because the sample is generated step by step, it is possible to start the test phase in parallel by outputting the products of the intermediate sample after each step. They compare IncLing against the sampling algorithms: Chvatal, IPOG, CASA on sample size, and runtime. Al-Hajjaji et al. conclude that IncLing does not show any drawbacks compared to the other algorithms in both regards. However, our evaluation shows that IncLing continuously computes samples that are a notch larger than compared to the other evaluated algorithms. We think the reason could be our model randomization used in our evaluation and the resulting randomized order of the input model's propositional formula.

Oh et al. [OBMS17] are the first to create true uniform random sampling for software product lines. Two case studies assess the usability of uniform random sampling for performance prediction and statistical information [PAP+19, PAMJ20]. Also, Oh et al. [OGB19a] use their uniform random sampling to compute t-wise samples. They measured evaluation criteria such as sample size, memory consumption, achieved t-wise coverage for $t = 1$ and $t = 2$, and sampling time. They conclude that uniform random sampling is not enough to achieve a 100% t-wise coverage.

Lei et al. [LKK+07] generalize an existing strategy, called In-Parameter-Order(IPO), from pairwise testing to t-way testing and named it IPOG. Their most significant challenge was the exponential growth of possible feature combinations. Krieter et al. [KTS+20] extend IPOG with several improvements. The resulting algorithm is named YASA, and his most distinctive property is that the user can configure the trade-off between sample size and sampling runtime. Their evaluation has a lot in common as they also compare YASA against Chvatal, ICPL, and IncLing. However, their evaluation compared the algorithms only on the sample size and sampling runtime. Their results show similarities to ours: YASA performs best for sample size and sample time, Chvatal is the slowest algorithm, and IncLing also continuously generates larger samples than the others.

In contrast to the work we have shown here, we are not introducing a new sampling algorithm. We have automated the evaluation and comparison of sampling algorithms. Also, we have carried out our empirical evaluation on a larger set of systems and for more criteria than the works mentioned above for a set of multiple sampling algorithms.

Comparing Existing Sampling Algorithms

Now we would like to focus exclusively on works that do not introduce new algorithms but only evaluate existing sampling algorithms.

Perrouin et al. [POS+12] compare CSP-dedicated and alloy-based approaches for pairwise sampling. Their evaluation focuses on comparing both approaches on the size of their generated samples and their runtime. In contrast to our evaluation,

https://doi.org/10.24355/dbbs.084-202009211318-0

they do not consider criteria like sample similarity and memory consumption during the sampling process.

Medeiros et al. [MKR+16] compare ten sampling strategies to investigate the trade-off between sample size and fault-detection capabilities (i.e., how many faults can be found in the included products). They conclude that samples with larger sizes also detect more errors and that some combinations of approaches provide a useful balance between sample size and fault-detection capabilities. In contrast to our evaluation, the authors do not consider criteria such as sample similarity, memory consumption, runtime, and t-wise interaction coverage. Further, we did not investigate the influence on the fault-detection capabilities as it requires the artifacts for all systems and a method to automatically detect the errors.

Halin et al. [HNA+19] performed a case-study to test all configurations of the industrial-strength open-source configurable software system JHipster. As part of their study, they compare the sample size and fault-detection-capabilities of nine different sampling strategies to investigate their impact on testing their system. In contrast to our evaluation, they do not consider other criteria besides the sample size and the fault/failure efficiency of their system.

https://doi.org/10.24355/dbbs.084-202009211318-0

9. Conclusion and Future Work

With our thesis, we aim to improve the reproducibility and credibility of t-wise product sampling algorithms. We identified great redundant efforts in the research area as authors manually compare sampling algorithms, which is time-consuming and, among other publications, hard to compare. We extend an existing survey on product sampling for software product lines by sixteen new publications. This extension shows that the research area is developing rapidly and that the comparison of most sampling algorithms is insufficient. We created a framework named AutoSamp (Automatic Sampling Framework), which automatically evaluates product sampling algorithms based on an available set of real-world models of various scales. AutoSamp produces easy comparable results, supporting researchers in assessing new and existing sampling algorithms. Another problem we identified is that industrial users lack the time and domain knowledge to compare all existing publications for sampling algorithms to select an appropriate one for their project. To address this problem, we created an evaluator named Tucs (T-Wise Use Case Scorer) that uses our framework's data to recommend a suitable sampling algorithm for the user while considering his requirements.

We designed AutoSamp to execute sampling algorithms while ensuring fairness by measuring evaluation criteria. We coupled AutoSamp with TIRA to provide all users the same machines without restricting their freedom. When a user finished their evaluation and is satisfied with their results, he can directly publish them via TIRA, making them directly available for a comparison to other sampling algorithms. We also introduced four approaches to score sampling algorithms based on the data provided by AutoSamp. The score of a sampling algorithm represents an automatically determined recommendation for the user while considering his requirements. We implemented all approaches in Tucs and integrated it into TIRA. So, that users could compute recommendations based on all available data for sampling algorithms on TIRA.

In our thesis, we performed an empirical evaluation with four sampling algorithms to determine the one that performs best for each evaluation criterion, i.e., the criteria used to compare sampling algorithms. Based on our evaluation, we concluded that

https://doi.org/10.24355/dbbs.084-202009211318-0

YASA performs best for runtime, memory consumption, and sample size. ICPL also proved as a viable alternative to compute small samples but requires more time and consumes more memory. We also concluded that ICPL computes the most similar samples while IncLing achieves the most dissimilar ones. We also evaluated our four score computation approaches to assess the correctness of their recommendations. We concluded that all approaches compute the correct recommendation when only considering one requirement. We concluded that the weighted rank-based score (WRBS) is the best approach for multiple requirements as it produces recommendations with 100% precision for all use-cases of our evaluation.

During our thesis, we identified multiple points that we can address in the future. First, our empirical evaluation only compares four sampling algorithms, and thus, we can evaluate a lot more sampling algorithms in the future. Second, our concept has some restrictions in TIRA as TIRA is still under development. Functionalities such as freely customizing evaluators, providing more data as hosts without the help of the TIRA team, and evaluating on all input runs would prove as viable additions to TIRA and our integration. Third, our evaluation for the score computation shows that our nominal-based score (NBS) produces bad recommendations as their normalization process uses bad transformation for some of the subscores. Therefore, selecting other transformations to achieve normalized values for the subscores could increase the recommendations' accuracy. Last but not least, our score computation evaluation only considers a set of use-cases that we defined ourselves. However, we think a survey to the community could help to identify the most required use cases that should be covered by an evaluation.

https://doi.org/10.24355/dbbs.084-202009211318-0

A. Appendix

https://doi.org/10.24355/dbbs.084-202009211318-0

Author	AlgorithmID	ModelID	ModelName	Model_Features	Model_Constraints	SystemIteration	AlgorithmIteration
Joshua Sprey	Chvatal_t2	0	BusyBox_2009-08-01_06-53-03	594	1837	1	1
Joshua Sprey	Chvatal_t2	0	BusyBox_2009-08-01_06-53-03	594	1837	2	1
Joshua Sprey	Chvatal_t2	0	BusyBox_2009-08-01_06-53-03	594	1837	3	1
Joshua Sprey	Chvatal_t2	0	BusyBox_2009-08-01_06-53-03	594	1837	4	1
Joshua Sprey	Chvatal_t2	0	BusyBox_2009-08-01_06-53-03	594	1837	5	1
Joshua Sprey	Chvatal_t2	0	BusyBox_2009-08-01_06-53-03	594	1837	6	1
Joshua Sprey	Chvatal_t2	0	BusyBox_2009-08-01_06-53-03	594	1837	7	1
Joshua Sprey	Chvatal_t2	0	BusyBox_2009-08-01_06-53-03	594	1837	8	1
Joshua Sprey	Chvatal_t2	0	BusyBox_2009-08-01_06-53-03	594	1837	9	1
Joshua Sprey	Chvatal_t2	0	BusyBox_2009-08-01_06-53-03	594	1837	10	1

Timeout	InTime	NoError	Time	Size	T-Value	Validity	Valid Conditions	Coverage	ROIC	MSOC	FIMD	ICST	Runtime	Throughput
86400000	true	true	132816	35	2	1	682494	1	-1	-1	-1	-1	128.98748	99.57578
86400000	true	true	140786	35	2	1	682494	1	-1	-1	-1	-1	136.56652	99.58655
86400000	true	true	139934	35	2	1	682494	1	-1	-1	-1	-1	136.04046	99.58084
86400000	true	true	145426	35	2	1	682494	1	-1	-1	-1	-1	142.89159	99.57352
86400000	true	true	139535	35	2	1	682494	1	-1	-1	-1	-1	135.41438	99.54594
86400000	true	true	140900	35	2	1	682494	1	-1	-1	-1	-1	136.84947	99.60156
86400000	true	true	135421	35	2	1	682494	1	-1	-1	-1	-1	131.61154	99.58285
86400000	true	true	137848	35	2	1	682494	1	-1	-1	-1	-1	133.68412	99.60019
86400000	true	true	134966	35	2	1	682494	1	-1	-1	-1	-1	133.98107	99.59367
86400000	true	true	143982	35	2	1	682494	1	-1	-1	-1	-1	141.0132	99.59649

TotalCreatedBytes	TotalPauseTime	AveragePauseTime
66913	0.5472	0.00855
66908	0.56463	0.00882
67108	0.57022	0.00891
67516	0.6094	0.01129
66933	0.61486	0.00976
67039	0.54527	0.00866
66869	0.54902	0.00871
66893	0.53448	0.00848
67921	0.54441	0.00864
67289	0.569	0.00889

Figure A.1: An example output table containing some framework evaluation results for the Chvatal algorithm

Model Date	#Features	#Constraints	Model Date	#Features	#Constraints
2007-05-20_17-12-43	439	463	2008-12-01_12-36-41	572	597
2007-06-01_14-40-03	439	463	2009-01-01_17-52-09	572	597
2007-07-01_14-53-06	454	480	2009-03-01_04-50-18	574	615
2007-08-01_23-30-54	459	486	2009-02-01_00-40-45	575	613
2007-09-02_14-51-54	469	501	2009-04-01_11-24-04	575	615
2007-10-01_09-59-01	472	505	2009-05-01_03-00-04	580	620
2007-11-02_23-31-10	475	511	2009-06-01_11-26-30	583	629
2007-12-02_01-43-18	482	515	2009-07-02_12-04-50	585	637
2008-01-02_19-55-04	487	521	2009-08-01_06-53-03	594	648
2008-02-01_01-41-57	493	525	2009-09-02_11-49-25	596	649
2008-04-01_14-47-57	535	562	2009-10-02_01-10-32	599	652
2008-05-02_09-19-29	537	566	2009-11-01_04-01-30	602	657
2008-06-01_10-10-22	543	586	2009-12-01_02-32-01	608	667
2008-03-01_09-35-39	546	582	2010-01-01_16-45-43	608	667
2008-09-01_15-23-04	546	601	2010-02-01_04-55-30	619	679
2008-07-01_01-57-36	550	593	2010-03-02_15-02-45	622	681
2008-10-02_13-30-31	556	576	2010-04-01_15-09-44	628	691
2008-08-01_02-15-05	557	612	2010-05-02_14-17-07	631	681
2008-11-01_00-10-51	561	581			

Table A.1: Overview of all information about the BusyBox feature models. The column *Model Date* shows the timestamp of the model in the following format: *yyyy-mm-dd_hh-mm-ss*. The column *#Features* shows the number of features while *#Constraints* shows the number of constraints

https://doi.org/10.24355/dbbs.084-202009211318-0

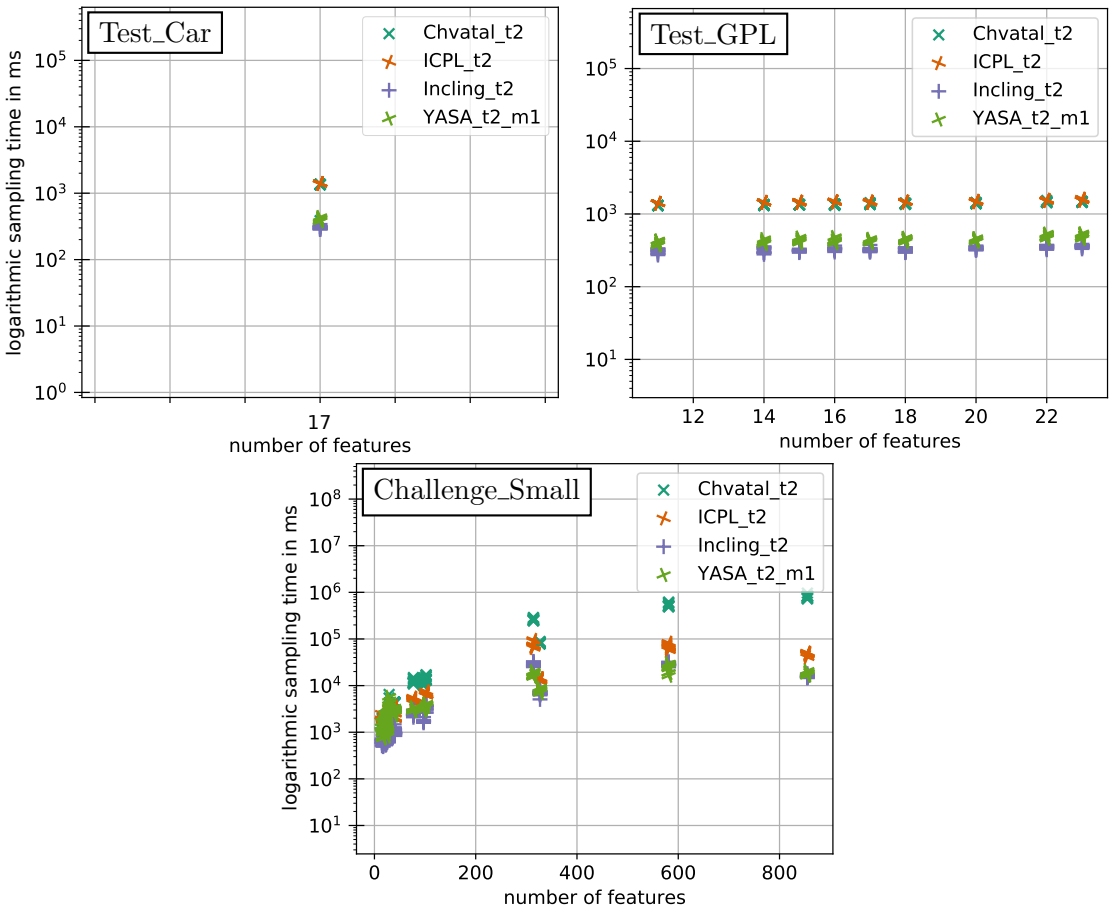


Figure A.2: Remaining evaluation results of the sampling time required to compute a sample for the skipped data packages (cf. see plot title). Calculations that exceeded the defined timeout or crashed due to errors are especially marked

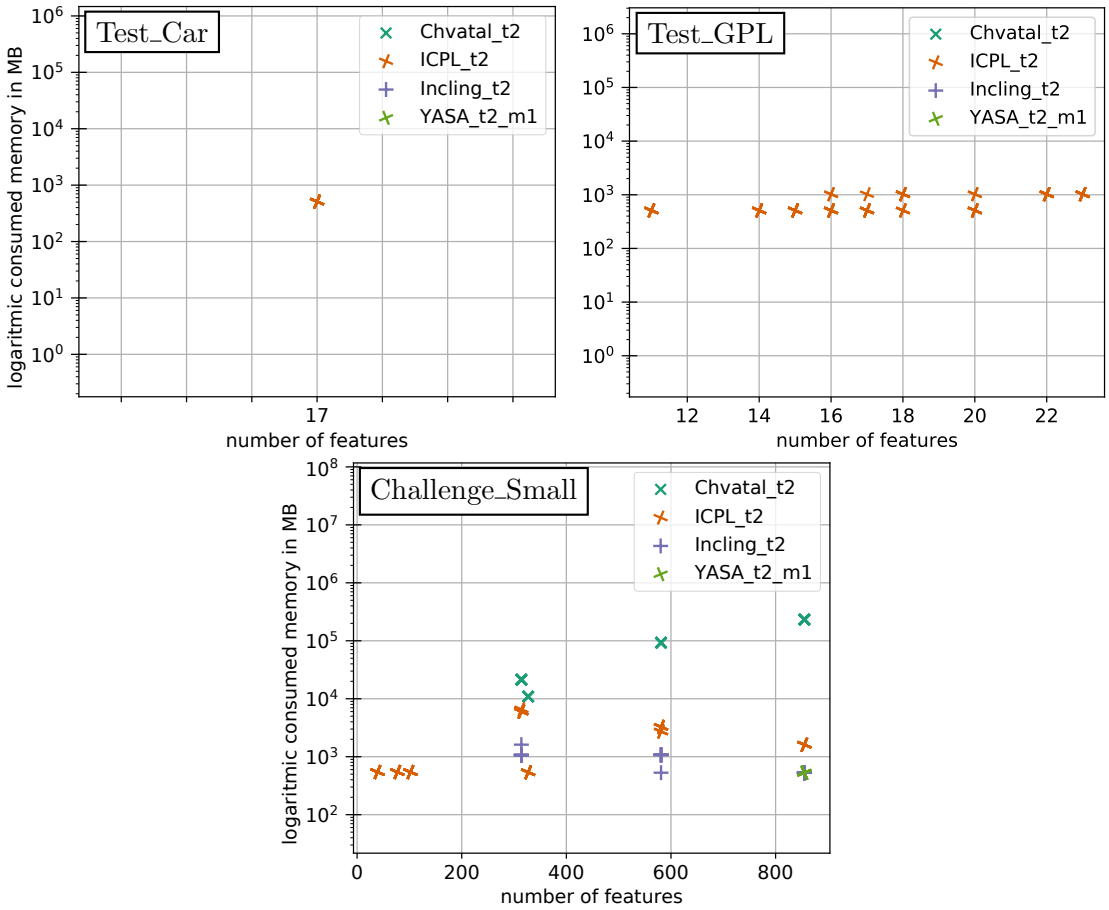


Figure A.3: Remaining evaluation results of the consumed memory for the sample process for the skipped data packages (cf. see plot title)

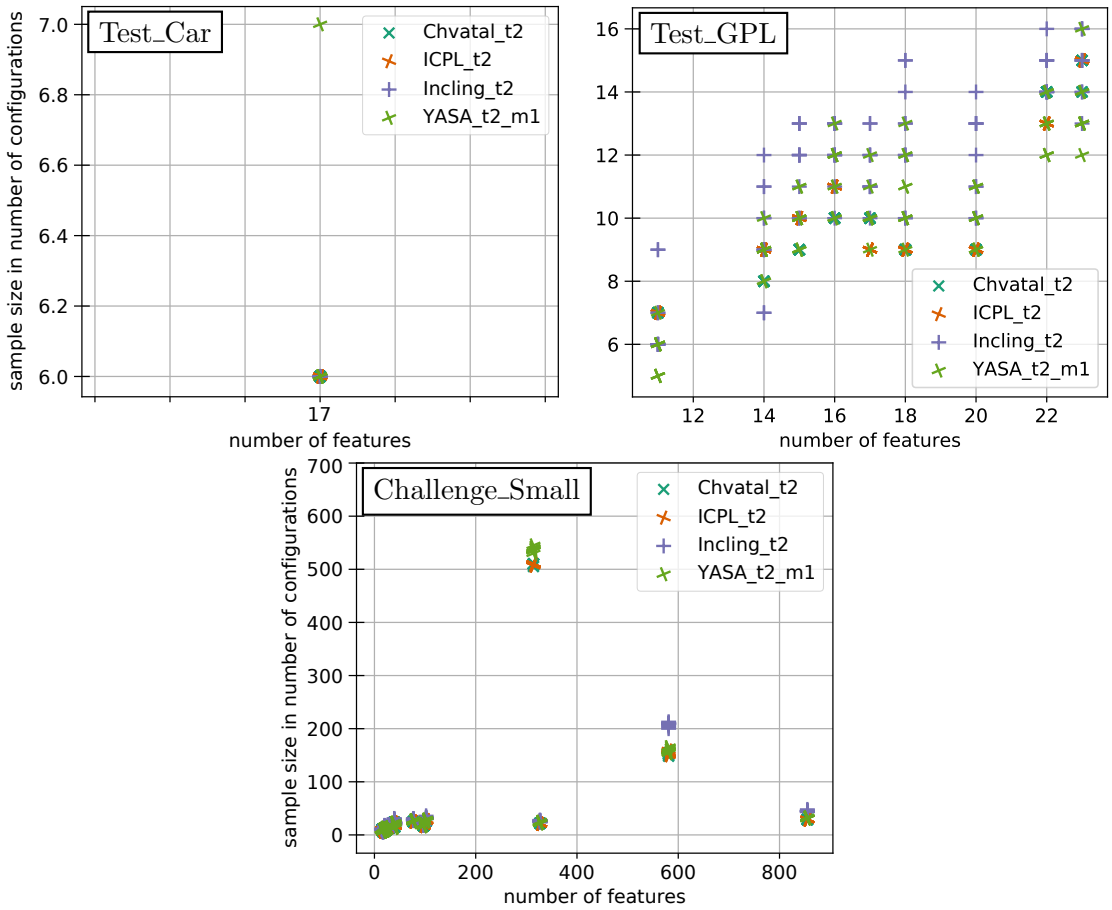


Figure A.4: Remaining evaluation results of the computed sample sizes for the skipped data packages (cf. see plot title)

https://doi.org/10.24355/dbbs.084-202009211318-0

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013. (cited on Page 1, 3, 4, 5, 6, 7, 8, 75, and 115)

[AGV15] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Generating Tests for Detecting Faults in Feature Models. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE Computer Science, April 2015. (cited on Page 21)

[AHKT+16] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. pages 144–155. ACM, October 2016. (cited on Page 11, 14, 17, 21, 75, and 91)

[AHLL+17] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. pages 34–40. IEEE Computer Science, May 2017. (cited on Page 22)

[AHMK+16] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. Tool Demo: Testing Configurable Systems with FeatureIDE. pages 173–177. ACM, October 2016. (cited on Page 21)

[AHTL+19] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective Product-Line Testing Using Similarity-Based Product Prioritization. 18(1):499–521, February 2019. (cited on Page 22)

[AMS+18] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):10:1–10:34, January 2018. (cited on Page 21)

[Ana16] Sofia Ananieva. *Explaining Defects and Identifying Dependencies in Interrelated Feature Models*. PhD thesis, Master's thesis. TU Braunschweig, Germany, 2016. (cited on Page 67)

https://doi.org/10.24355/dbbs.084-202009211318-0

[AvRW+13] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 482–491. IEEE Computer Science, May 2013. (cited on Page 115)

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20. Springer, 2005. (cited on Page 32)

[BGH+06] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006. (cited on Page 86)

[BL14] Hauke Baller and Malte Lochau. Towards Incremental Test Suite Optimization for Software Product Lines. In *Proceedings of the International SPLC Workshop Feature-Oriented Software Development (FOSD)*, pages 30–36. ACM, 2014. (cited on Page 21)

[BLLS14] Hauke Baller, Sascha Lity, Malte Lochau, and Ina Schaefer. Multi-Objective Test Suite Optimization for Incremental Product Family Testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 303–312. IEEE Computer Science, 2014. (cited on Page 21)

[BMB+11] Marko Bošković, Gunter Mussbacher, Ebrahim Bagheri, Daniel Amyot, Dragan Gašević, and Marek Hatala. Aspect-Oriented Feature Models. In *Proceedings of the International Conference on Models in Software Engineering (MODELSWARD)*, pages 110–124. Springer, 2011. (cited on Page 67)

[BRCTS06] David Benavides, Antonio Ruiz-Cortés, Pablo Trinidad, and Sergio Segura. A Survey on the Automated Analyses of Feture Models. *Jornadas de Ingeniera del Software y Bases de Datos*, pages 367–376, 2006. (cited on Page 7)

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 7:1–7:8. ACM, 2013. (cited on Page 3, 35, and 115)

[BSL+12] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. Variability modeling in the systems software domain. *Generative Software Development Laboratory, University of Waterloo, Technical Report*, 2012. (cited on Page 67 and 68)

https://doi.org/10.24355/dbbs.084-202009211318-0

[BSL+13] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering (TSE)*, 39(12):1611–1640, 2013. (cited on Page 68 and 69)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010. (cited on Page 5 and 6)

[Bus] *BusyBox*. Accessed: July 14, 2020. (cited on Page 69)

[CDFP97] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997. (cited on Page 1, 90, and 91)

[CDS08] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering (TSE)*, 34(5):633–650, 2008. (cited on Page 21 and 23)

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, 2000. (cited on Page 3 and 4)

[Cha93] DIMACS Challenge. Satisfiability: Suggested format. *DIMACS Challenge. DIMACS*, 1993. (cited on Page 32)

[Chv79] Vasek Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of operations research*, 4(3):233–235, 1979. (cited on Page 10, 14, 16, and 90)

[CMMCDA14] Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. 56(10):1183–1199, 2014. (cited on Page 1, 2, 9, and 90)

[CMMDA12] Ivan Do Carmo Machado, John D. McGregor, and Eduardo Santana De Almeida. Strategies for Testing Products in Software Product Lines. *Software Engineering Notes (SEN)*, 37(6):1–8, November 2012. (cited on Page 1)

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001. (cited on Page 3 and 4)

[CR14] Anastasia Cmyrev and Ralf Reissing. Efficient and Effective Testing of Automotive Software Product Lines. 7(2), 2014. (cited on Page 21)

https://doi.org/10.24355/dbbs.084-202009211318-0

[DMTR08] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 71–82, 2008. (cited on Page 1 and 65)

[DPL+15] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Covering SPL Behaviour with Sampled Configurations: An Initial Assessment. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 59:59–59:66. ACM, 2015. (cited on Page 22 and 86)

[EBA+11] Alireza Ensan, Ebrahim Bagheri, Mohsen Asadi, Dragan Gasevic, and Yevgen Biletskiy. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *Proceedings of the International Conference on Information Technology:New Generations (ITNG)*, pages 291–298. IEEE Computer Science, 2011. (cited on Page 21)

[EBG12] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gasevic. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 7328, pages 613–628. Springer, 2012. (cited on Page 21)

[FCP09] Sandro Fouché, Myra B Cohen, and Adam Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 177–188, 2009. (cited on Page 9)

[FD12] Cool Features and Tough Decisions. A comparison of variability modeling approaches. *Krzysztof Czarnecki et al*, 2012. (cited on Page 3)

[Fea20] FeatureIDE. *FeatureIDE | An extensible framework for feature-oriented software development*, 2017 (accessed June 21, 2020). (cited on Page 49)

[FKPV16] Thiago N. Ferreira, Josiel Neumann Kuk, Aurora Pozo, and Silvia Regina Vergilio. Product Selection Based on Upper Confidence Bound MOEA/D-DRA for Testing Software Product Lines. pages 4135–4142. IEEE Computer Science, July 2016. (cited on Page 21)

[FLHRE16] Stefan Fischer, Roberto E. Lopez-Herrejon, Rudolf Ramler, and Alexander Egyed. A Preliminary Empirical Assessment of Similarity for Combinatorial Interaction Testing of Software Product Lines. pages 15–18. ACM, 2016. (cited on Page 22)

https://doi.org/10.24355/dbbs.084-202009211318-0

[FLS+17] Thiago N. Ferreira, Jackson A. Prado Lima, Andrei Strickler, Josiel N. Kuk, Silvia R. Vergilio, and Aurora Pozo. Hyper-Heuristic Based Product Selection for Software Product Line Testing. 12(2):34–45, May 2017. (cited on Page 21)

[FLV17] Helson L. Jakubovski Filho, Jackson A. Prado Lima, and Silvia R. Vergilio. Automatic Generation of Search-Based Algorithms Applied to the Feature Testing of Software Product Lines. pages 114–123. ACM, 2017. (cited on Page 21)

[GCD11] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing. *Empirical Software Engineering (EMSE)*, 16(1):61–102, 2011. (cited on Page 14, 21, and 90)

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. (cited on Page 54)

[GKS+14] Alexander Grebhahn, Sebastian Kuckuk, Christian Schmitt, Harald Köstler, Norbert Siegmund, Sven Apel, Frank Hannig, and Jürgen Teich. Experiments on Optimizing the Performance of Stencil Codes with SPL Conqueror. 24(3), 2014. (cited on Page 21)

[GRS+17] Alexander Grebhahn, Carmen Rodrigo, Norbert Siegmund, Francisco José Gaspar, and Sven Apel. Performance-Influence Models of Multigrid Methods: A Case Study on Triangular Grids. 29(17), 2017. (cited on Page 22)

[GYS+18] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wąsowski, and Huiqun Yu. Data-Efficient Performance Learning for Configurable Systems. *Empirical Software Engineering (EMSE)*, 23(3):1826–1867, June 2018. (cited on Page 21)

[HB10] Hadi Hemmati and Lionel Briand. An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 141–150. IEEE Computer Science, November 2010. (cited on Page 39 and 78)

[HLHE13] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 16:1–16:6. ACM, 2013. (cited on Page 21)

[HMB+15] Allan Hanbury, Henning Müller, Krisztian Balog, Torben Brodt, Gordon V Cormack, Ivan Eggel, Tim Gollub, Frank Hopfgartner,

https://doi.org/10.24355/dbbs.084-202009211318-0

Jayashree Kalpathy-Cramer, Noriko Kando, et al. Evaluation-as-a-service: Overview and outlook. *arXiv preprint arXiv:1512.07454*, 2015. (cited on Page 89)

[HMGB16] Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, and Benoit Baudry. Practical minimization of pairwise-covering test configurations using constraint programming. *Information and Software Technology*, 71:129–146, 2016. (cited on Page 86)

[HNA+19] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering*, 24(2):674–717, July 2019. (cited on Page 8, 22, 90, and 92)

[HPLT14] Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-Based Generation of Software Product Line Test Configurations. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer International Publishing, 2014. (cited on Page 21)

[HPP+12] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Suites for Large Software Product Lines. *CoRR*, abs/1211.5451, 2012. (cited on Page 8)

[HPP+13] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-Objective Test Generation for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 62–71. ACM, 2013. (cited on Page 21, 71, and 83)

[HPP+14] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Transactions on Software Engineering (TSE)*, 40(7):650–670, July 2014. (cited on Page 6, 14, 21, 27, and 86)

[JHF11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. pages 638–652. Springer, 2011. (cited on Page 10, 21, and 90)

[JHF12a] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 46–55. ACM, 2012. (cited on Page 1, 8, 9, 10, 11, 14, 16, 21, and 90)

https://doi.org/10.24355/dbbs.084-202009211318-0

[JHF+12b] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. pages 269–284. Springer, 2012. (cited on Page 21)

[KAB07] Christian Kästner, Sven Apel, and Don Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 223–232. IEEE Computer Science, 2007. (cited on Page 67)

[KAT16a] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining Anomalies in Feature Models. pages 132–143. ACM, October 2016. (cited on Page 5)

[KAT16b] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining Anomalies in Feature Models. Technical Report 2016-01, TU Braunschweig, Germany, August 2016. (cited on Page 67)

[KBBK10] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. Reducing Configurations to Monitor in a Software Product Line. pages 285–299. Springer, November 2010. (cited on Page 21)

[KBK11] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 57—68. ACM, 2011. (cited on Page 21)

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (cited on Page 1 and 3)

[KGS+19] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. Distance-Based Sampling of Software Configuration Spaces. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1084–1094. IEEE Computer Science, 2019. (cited on Page 8, 9, 21, 23, and 90)

[KLD02] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. 19(4):58–65, 2002. (cited on Page 1)

[KPK+17] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. FeatureIDE: Empowering Third-Party Developers. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 42–45. ACM, September 2017. (cited on Page 49 and 85)

https://doi.org/10.24355/dbbs.084-202009211318-0

[KSS13] Matthias Kowal, Sandro Schulze, and Ina Schaefer. Towards Efficient SPL Testing by Variant Reduction. In *Conference of the International workshop on Variability & composition (VariComp)*, pages 1–6. ACM, 2013. (cited on Page 21)

[KTM+18] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is There a Mismatch between Real-World Feature Models and Product-Line Research? pages 53–54, March 2018. (cited on Page 3, 68, and 85)

[KTS+19] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. Propagating Configuration Decisions with Modal Implication Graphs. pages 77–78, February 2019. (cited on Page 69)

[KTS+20] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. YASA: Yet Another Sampling Algorithm. ACM, February 2020. (cited on Page 11, 14, 16, 21, 23, 75, 86, 90, and 91)

[LCS+16] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your {JVM}: Understand and eliminate {JVM} warm-up overhead in data-parallel systems. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 383–400, 2016. (cited on Page 86)

[LFC+14] Roberto Erick Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. Comparative Analysis of Classical Multi-Objective Evolutionary Algorithms and Seeding Strategies for Pairwise Testing of Software Product Lines. pages 387–396. IEEE Computer Science, 2014. (cited on Page 22)

[LGL19] Lars Luthmann, Timo Gerecht, and Malte Lochau. Sampling Strategies for Product Lines with Unbounded Parametric Real-Time Constraints. 21(6):613–633, 2019. (cited on Page 21)

[LKK+07] Yu Lei, Raghu N. Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG: A General Strategy for T-Way Software Testing. In *Proceedings of the International Conference on Engineering of Computer-Based Systems (ECBS)*, pages 549–556. IEEE Computer Science, 2007. (cited on Page 14, 90, and 91)

[LM06] Jaejoon Lee and Dirk Muthig. Feature-oriented variability management in product line engineering. *Communications of the ACM*, 49(12):55–59, 2006. (cited on Page 1)

[Lut13] Peter Lutz. *Typprüfung von Produktlinien in Fuji*. PhD thesis, 2013. (cited on Page 67)

https://doi.org/10.24355/dbbs.084-202009211318-0

[LvRK+13] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, pages 81–91. ACM, August 2013. (cited on Page 21)

[Man02] Mike Mannion. Using first-order logic for product line model validation. In *International Conference on Software Product Lines*, pages 176–187. Springer, 2002. (cited on Page 7)

[McG01] John McGregor. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022, Carnegie Mellon University, 2001. (cited on Page 1, 8, and 65)

[MFBW16] Jean Melo, Elvis Flesborg, Claus Brabrand, and Andrzej Wąsowski. A quantitative analysis of variability warnings in linux. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, pages 3–8, 2016. (cited on Page 11 and 27)

[MFV16] Rui Angelo Matnei Filho and Silvia Regina Vergilio. A Multi-Objective Test Data Generation Approach for Mutation Testing of Feature Models. 4(1), July 2016. (cited on Page 21)

[MGSH13] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical Pairwise Testing for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 227–235. ACM, 2013. (cited on Page 21)

[MKR+16] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 643–654. ACM, 2016. (cited on Page 1, 8, 21, 27, 65, and 92)

[MNM+18] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 155–166, 2018. (cited on Page 35)

[MOP+19] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 289–301. ACM, 2019. (cited on Page 21)

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017. (cited on Page 8, 16, 32, and 67)

https://doi.org/10.24355/dbbs.084-202009211318-0

[NMS+18] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. Anomaly Analyses for Feature-Model Evolution. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 188–201. ACM, November 2018. (cited on Page 5 and 69)

[OBMS17] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 61–71, August 2017. (cited on Page 21, 23, 90, and 91)

[OGB19a] Jeho Oh, Paul Gazzillo, and Don Batory. t-wise Coverage by Uniform Sampling. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 84–87. ACM, September 2019. (cited on Page 8, 22, 23, 90, and 91)

[OGB+19b] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. Uniform Sampling from Kconfig Feature Models. Technical Report TR-19-02, The University of Texas at Austin, Department of Computer Science, 2019. (cited on Page 8, 21, 23, and 90)

[OMR10] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 196–210. Springer, 2010. (cited on Page 2, 9, 14, 17, 21, and 90)

[OZLG11] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 6:1–6:8. ACM, 2011. (cited on Page 22)

[PAMJ20] Juliana Alves Pereira, Mathieu Acher, Hugo Martib, and Jean-Marc Jézéquel. Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. pages 277–288. ACM, 2020. (cited on Page 9, 22, 23, 90, and 91)

[PAP+19] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet? In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 240–251. IEEE Computer Science, 2019. (cited on Page 8, 22, 23, 90, and 91)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, September 2005. (cited on Page 3 and 4)

[Pet18a] Tobias Pett. *Stability of Product Sampling under Product-Line Evolution.* PhD thesis, Master's thesis. TU Braunschweig, Germany, 2018. (cited on Page 20, 39, 55, and 78)

https://doi.org/10.24355/dbbs.084-202009211318-0

[Pet18b] Tobias Pett. Stability of Product Sampling under Product-Line Evolution. Master's thesis, TU Braunschweig, Germany, November 2018. (cited on Page 69)

[PGWS19] Martin Potthast, Tim Gollub, Matti Wiegmann, and Benno Stein. Tira integrated research architecture. In *Information Retrieval Evaluation in a Changing World*, pages 123–160. Springer, 2019. (cited on Page 13, 28, 29, 30, 31, 49, 89, and 115)

[Ple18] Hans E Plesser. Reproducibility vs. replicability: a brief history of a confused terminology. *Frontiers in neuroinformatics*, 11:76, 2018. (cited on Page 89)

[POS+12] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journa l (SQJ)*, 20(3-4):605–643, 2012. (cited on Page 1, 9, 22, 90, and 91)

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 459–468. IEEE Computer Science, April 2010. (cited on Page 21 and 90)

[PTR+19] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. Product Sampling for Product Lines: The Scalability Challenge. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 78–83. ACM, September 2019. (cited on Page 22, 32, 69, and 116)

[PYW11] Jörg Pleumann, Omry Yadan, and Erik Wetterberg. Antenna: An Ant-to-End Solution For Wireless Java. Website, 2011. Available online at http://antenna.sourceforge.net/; visited on November 22nd, 2011. (cited on Page 8)

[RBR+15] Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 131–140. ACM, 2015. (cited on Page 21)

[Res17] ACM Result. Artifact review and badging, 2017. (cited on Page 89)

[RLB+18] Sebastian Ruland, Lars Luthmann, Johannes Bürdek, Sascha Lity, Thomas Thüm, Malte Lochau, and Márcio Ribeiro. Measuring Effectiveness of Sample-Based Product-Line Testing. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 119–133. ACM, November 2018. (cited on Page 22)

https://doi.org/10.24355/dbbs.084-202009211318-0

[SCD12] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 270–284. Springer, March 2012. (cited on Page 21)

[SGAK15] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, pages 284–294. ACM, August 2015. (cited on Page 22)

[SGS+15] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-Efficient Sampling for Performance Prediction of Configurable Systems. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 342–352. IEEE Computer Science, 2015. (cited on Page 21)

[SKK+12] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting Performance via Automated Feature-Interaction Detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 167–177. IEEE Computer Science, 2012. (cited on Page 21)

[SRK+12] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journa l (SQJ)*, 20(3-4):487–517, September 2012. (cited on Page 21)

[SRK+13] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption. 55(3):491–507, March 2013. (cited on Page 21)

[Sto10] Victoria Stodden. The scientific method in practice: Reproducibility in the computational sciences. 2010. (cited on Page 89)

[STS20] Chico Sundermann, Thomas Thüm, and Ina Schaefer. Evaluating #SAT Solvers on Industrial Feature Models. ACM, February 2020. (cited on Page 69)

[SvRA13] Norbert Siegmund, Alexander von Rhein, and Sven Apel. Family-Based Performance Measurement. pages 95–104. ACM, 2013. (cited on Page 22)

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for

https://doi.org/10.24355/dbbs.084-202009211318-0

Software Product Lines. 47(1):6:1–6:45, June 2014. (cited on Page 90)

[TBD06] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 191–200, 2006. (cited on Page 67)

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE Computer Science, May 2009. (cited on Page 39)

[TDS+14] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. pages 421–432, Berkeley, CA, USA, 2014. USENIX Association. (cited on Page 21)

[TIR20] TIRA. *TIRA | Task Market*, 2012 (accessed June 22, 2020). (cited on Page 50)

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85, January 2014. (cited on Page 85)

[TLD+12] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, January 2012. (cited on Page 19 and 21)

[VAHT+18] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A Classification of Product Sampling for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 1–13. ACM, September 2018. (cited on Page 1, 2, 8, 15, 17, 18, 20, 21, 22, 23, 27, 35, 90, 115, and 116)

[vdML04] Thomas von der Maßen and Horst Lichter. Deficiencies in feature models. In *workshop on software variability management for product derivation-towards tool support*, volume 44, page 21, 2004. (cited on Page 6)

[vRAK+13] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. The PLA Model: On the Combination of Product-Line Analyses. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 14:1–14:8. ACM, January 2013. (cited on Page 90 and 115)

https://doi.org/10.24355/dbbs.084-202009211318-0

[Wei08] David M. Weiss. The Product Line Hall of Fame. In *Proceedings of the International Software Product Line Conference (SPLC)*, page 395. IEEE Computer Science, 2008. (cited on Page 1)

https://doi.org/10.24355/dbbs.084-202009211318-0

Topic Description

Configurable systems proved as an economic and successful approach to develop tailored products for customers [ABKS13]. Benefits such as individual products, reduced development costs, and faster time-to-market are essential reasons for the industry to develop highly configurable systems [BRN+13]. However, quality assurance of such systems remains a challenging task as the interactions between configuration options can reveal bugs, and, thus, they need to be tested. Naively, this includes testing every product of the system which is manually infeasible as there are too many products [AvRW+13].

A possible trade-off between testing effort and quality assurance is *product sampling*, which aims to test only a subset of all products (e.g., a sample), while achieving a certain coverage [VAHT+18]. However, finding suitable samples is a challenging problem itself. Therefore, researchers strive to improve sampling algorithms continuously [VAHT+18, vRAK+13]. Comparing these techniques can be quite difficult as their software and data are not always publicly available and often focus on different requirements (e.g., feature-interaction coverage or code coverage [VAHT+18]) [PGWS19]. The ability to compare sampling techniques automatically saves effort as developers doesn't need to reconstruct unavailable software and data themselves. Furthermore, it can help users deciding on the most appropriate technique for their system based on *evaluation criteria*. These criteria can be used to reason about the quality of a sampling algorithm for a certain area of application. Evaluation criteria are categorized into *sampling efficiency* (e.g., runtime of the sampling algorithm), *testing efficiency* (e.g., size of sample), and *testing effectiveness* (e.g., t-wise coverage) [VAHT+18].

The goal of the master thesis is to help users decide on an appropriate sampling algorithm for their system based on the user's evaluation criteria. The selection of an appropriate sampling algorithm should be performed by a platform that automatically compares sampling algorithms and sorts the result based on the user's emphasis (e.g., the size of the sample is double as important as the runtime of the algorithm).

Such a platform for comparing sampling algorithms may be possible to realize using TIRA[1] Integrated Research Architecture, a modularized platform with the aim to improve the reproducibility of shared tasks in computer science [PGWS19].

[1] https://www.tira.io/

https://doi.org/10.24355/dbbs.084-202009211318-0

Task 1 (Concept)

- Update the survey [VAHT+18] of the different feature-model-based sampling algorithms and their evaluation criteria.
- Propose how to select the most appropriate algorithm when two or more evaluation criteria are desired (e.g., Pareto optimality).
- Determine whether TIRA can be used to realize the automated platform. If not find alternatives.
- Consider that the platform will be used for a challenge[PTR+19] in the future where developers can submit their own models/techniques. Therefore, a standardized input and output format should be used for all internal processes/algorithms.

Task 2 (Implementation)

Implement your platform according to your concept.

Task 3 (Evaluation)

- Perform a replication study of sampling algorithms using your platform and show that it works as intended.
- Perform an evaluation of your Pareto optimal selection of sampling algorithms.

Supervision

The thesis is supervised by Dr.-Ing. Thomas Thüm, Tobias Pett, and Prof. Dr.-Ing. Ina Schaefer, Institute of Software Engineering and Automotive Informatics.

Braunschweig, den 04. August 2020

https://doi.org/10.24355/dbbs.084-202009211318-0

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Braunschweig, den 04. August 2020

https://doi.org/10.24355/dbbs.084-202009211318-0